Rene Kultz

Utilização de Heurísticas de Planejamento no desdobramento de redes de Petri

Curitiba - PR

02/06/2010

Rene Kultz

Utilização de Heurísticas de Planejamento no desdobramento de redes de Petri

Dissertação apresentada como requisito parcial à obteção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador:
Prof. Dr. Luis Allan Künzle

Co-orientador:
Prof. Dr. Fabiano Silva

Universidade Federal do Paraná - UFPR

Curitiba - PR

02/06/2010

AGRADECIMENTOS

Ao meu Senhor, Jesus Cristo, pelo alto preço pago na cruz, pelo dom da vida e por todo o suporte e direção recebidos ao longo de minha vida, sem os quais a conclusão deste curso seria impossível. A Ele, toda a glória.

Ao meu orientador, Prof. Dr. Luis Allan Künzle, por conduzir o trabalho com toda dedicação e paciência.

Ao meu co-orientador, Prof. Dr. Fabiano Silva, por igualmente ter auxiliado através de sugestões preciosas em momentos chaves da pesquisa.

À minha irmã, Jaqueline Kultz e ao meu sobrinho, Samuel Kultz Alves, por terem me acolhido em sua casa e me tratado com todo carinho e compreensão durante o período do mestrado.

Aos meus pais, Ailton Kultz e Sirlei Paulina Barbosa Kultz, por terem acreditado e investido em mim de todas as maneiras possíveis, me dando as ferramentas através das quais pude conquistar meus sonhos.

Aos meus amigos que conquistei em Curitiba, por terem me acolhido e me dado apoio, tornando-se uma segunda família.

Abstract

Many works involve the relation between the problems of Classical Planning and reachability in Petri nets, due the proximity existing between these two formalisms. One of the techniques that produce better results in the reachability solution is known as “unfolding”. The resulting net from the unfolding process has size exponentially bigger than the size of the initial net, though this net is smaller than the size of the reachability graph. This work aims to adapt the planning heuristics H^1 and H^2, based on the regression from a goal state, to guide the construction of the unfolding until the goal mark can be reached, allowing that the solution can be extracted from the occurrence net generated. The adaptation was done through a data structure called *vector calculus*, which enumerates all the subsets with size smaller or equals to *m*, according the heuristic order, allowing some optimizations in the heuristic calculus. Experimental results were obtained from Petri nets generated from the planner *Petrigraph*, which converts classic planning problems from the format PDDL to Petri Nets. These nets were submited to unfolding using the heuristics H^1 and H^2, and the results were compared with the heuristic implemented by Töws and with the planner *SatPlan*. Besides, analysis were made involving the number of expansions done until the solution be found, the work realized by the tool *Mole*, the complexity from the vector calculus and the depth reached by the nets where the solution was not found.

Resumo

Diversos trabalhos envolvem a relação existente entre os problemas de Planejamento Clássico e os problemas de alcançabilidade de redes de Petri, em virtude da proximidade existente entre estes dois formalismos. Uma das técnicas que produz melhores resultados na solução de problemas de alcançabilidade é conhecida como "Desdobramento". A rede resultante do desdobramento possui complexidade exponencial em relação ao tamanho inicial da rede de Petri, ainda que produza uma rede menor do que o tamanho do grafo de alcançabilidade de redes de Petri. O objetivo deste trabalho é adaptar as heurísticas de Planejamento H^1 e H^2, baseadas na regressão de um estado objetivo, para guiar o processo de desdobramento da rede de Petri até que seja atingida uma marcação objetivo, permitindo que a solução possa ser extraída da rede de ocorrências gerada. Esta adaptação foi feita a partir de uma estrutura de dados chamada de *vetor de cálculo*, que enumera as regressões de todos os subconjuntos de tamanho menor ou igual a *m*, de acordo com a ordem da heurística, permitindo algumas otimizações no cálculo da heurística. Resultados experimentais foram obtidos a partir de redes de Petri geradas a partir do planejador *Petrigraph*, que converte problemas de planejamento clássico descritos em forma *PDDL* em forma de redes de Petri. Estas redes foram submetidas ao desdobramento com auxílio das heurísticas H^1 e H^2, sendo os resultados comparados com a heurística implementada por Töws e com o planejador *SatPlan*. Também foram feitas análises envolvendo o número de expansões realizadas até ser encontrada a solução, o trabalho total realizado pela ferramenta *Mole*, a complexidade do vetor de cálculo e a profundidade atingida nas redes em que a solução não foi encontrada.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

1 Introdução

Este trabalho foi desenvolvido em continuidade de pesquisas envolvendo planejamento e redes de Petri, realizadas pelo grupo de pesquisas LIAMF (Laboratório de Inteligência Artificial e Métodos Formais), da Universidade Federal do Paraná. Dentre as pesquisas realizadas por este grupo, destacam-se os trabalhos envolvendo a relação entre Planejamento e redes de Petri. O ambiente de Planejamento envolve problemas em que, dado um estado inicial, busca-se uma sequência ordenada de ações capazes de atinir um objetivo [24]. Engloba problemas em que se prevê situações novas, parâmetros ou objetivos complexos, ou quando ocorre em ambientes envolvendo altos riscos ou altos custos, atividades conjuntas com outras pessoas ou atividades sincronizadas com um sistema dinâmico [9]. As redes de Petri constituem-se em uma ferramenta matemática e gráfica de modelagem, que atraiu a atenção de diversos pesquisaores por possuir mecanismos de análise poderosos, permitindo a verificação de propriedades e da corretude do sistema especificado [23]. Uma rede de Petri pode modelar sistemas paralelos, concorrentes, assíncronos e não-determinísticos, se adaptando bem a aplicações nas quais noções de eventos e evoluções são importantes. As redes de Petri possuem características comuns ao ambiente de planejamento, especialmente com o Planejamento Clássico, que engloba ambientes completamente observáveis, determinísticos, finitos, estáticos e discretos [24]. Tanto as redes de Petri quanto o Planejamento Clássico são caracterizados pela definição de estados através de um conjunto de variáveis de estado binárias e a modificação de um estado é realizada através de um conjunto de ações. Os problemas de alcançabilidade de redes de Petri, cujo objetivo é descobrir se uma determinada marcação é alcançável a partir da marcação inicial, possuem um objetivo parecido com o objetivo da busca realizada pelos problemas de Planejamento. Esta proximidade motivou diversos trabalhos envolvendo os dois formalismos.

Em [29], foi proposta uma técnica de conversão de problemas de planejamento para problemas de alcançabilidade de redes de Petri, chamada de *Petrigraph*. O Petrigraph é um aperfeiçoamento do planejador *PUP*, proposto por Hickmott e colegas em [12]. O *PUP* é um planejador que, dada uma entrada composta por um problema e um domínio de planejamento, no formato *PDDL* [18], produz uma rede de Petri cíclica equivalente ao problema de planejamento. A rede gerada pelo *PUP* possui exatamente o dobro de variáveis em relação ao número de variáveis de estados do problema de planejamento original, enquanto que a proporção de transições em relação ao número de ações do problema de planejamento tende a ser bem maior.

No caso do *Petrigraph*, parte da conversão de problemas de planejamento em redes de Petri é feita com a identificação de grupos de variáveis multivaloradas no problema de planejamento. Os grupos de variáveis multivaloradas são aqueles em que, em qualquer estado alcançável do problema, apenas uma das variáveis é verdadeira. A identificação destes grupos de variáveis no problema de planejamento permite que a conversão para redes de Petri seja feita de uma maneira mais eficiente, reduzindo o tamanho da rede de Petri em relação ao tamanho das redes geradas pelo *PUP*, sem que ocorra perda de informação.

As redes de Petri geradas a partir do *PUP* e do *Petrigraph* descrevem o problema de planejamento, mas ainda precisam ser submetidas a um processo de busca a fim de que o plano possa ser extraído da rede. O algoritmo usado no processo de busca é o algoritmo de desdobramento de redes de Petri (*unfolding*) [7]. Este algoritmo cria, a partir da rede de Petri de origem, uma outra rede de Petri chamada de *Rede de Ocorrências*, cuja função é explorar o espaço de estados da rede de Petri de uma maneira mais eficiente do que outras técnicas utilizadas. A rede de ocorrências possui a característica de ser acíclica e de evitar conflitos envolvendo mais de dois elementos. A condição de parada do algoritmo de desdobramento é a obtenção do *prefixo completo* da rede, que é a configuração que contém todas as marcações possíveis de serem alcançadas a partir do estado inicial da rede de Petri. No caso das redes de Petri geradas pelo *PUP* e pelo *Petrigraph*, a utilização do algoritmo de desbramento não possui o objetivo de obter o prefixo completo, mas apenas uma parte da rede de ocorrências que contenha a solução. Por esta razão, o algoritmo de desdobramento foi modificado para ser executado conforme a lógica de algoritmos de busca clássicos de inteligência artificial, sendo auxiliado com a heurística de planejamento H^1, que é a variação mais simples das heurísticas da família H^m, propostas em [11]. A implementação da heurística H^1 por Hickmott [12] para a realização da busca pelo *PUP* foi realizada com auxílio da ferramenta *Mole* [25], criada para realizar o desdobramento de redes de Petri seguras. A versão modificada do *Mole* não foi disponibilizada, o que impediu a verificação da validade da mesma. Por outro lado, o *Petrigraph* pertence ao mesmo grupo de pesquisa a que a presente dissertação faz parte, o que permitiu uma análise do código-fonte da versão modificada do *Mole* para acoplar a heurística H^1. Esta análise revelou diferenças significativas entre a heurística efetivamente implementada e a heurística H^1 proposta em [11]. Além disso, a maneira como a heurística foi implementada não permite que sejam utilizadas heurísticas de ordem maior do que 1.

Neste trabalho, foi proposta uma adaptação alternativa das heurísticas de planejamento da família H^m no desdobramento de redes de Petri, a fim de auxíliar na solução de problemas de alcançabilidade de redes de Petri. Os problemas de alcançabilidade são tratados neste trabalho não apenas como problemas de decisão, onde a ferramenta se limita a verificar se determinada marcação é alcançável ou não a partir da marcação inicial, mas também se preocupa em retornar um plano composto por uma sequencia ordenada de transições que, disparadas em sequencia, transformam o estado inicial da rede no estado objetivo. A definição do estado objetivo é feita

na própria rede de Petri alvo do desdobramento, com a inserção de uma transição objetivo cujas pré-condições correspondem à marcação objetivo da rede. Uma vez que a transição objetivo for disparada, o algoritmo de desdobramento é encerrado e o plano ótimo pode ser extraído diretamente da rede de ocorrências gerada até o momento. A ferramenta *Mole* foi modificada para realizar o desdobramento conforme a lógica de funcionamento do algoritmo de busca A*, utilizando as heurísticas H^1 e H^2, provenientes da família de heurísticas H^m. Como as heurísticas da família H^m são admissíveis e monotônicas, garante-se a obtenção do plano ótimo. A adaptação das heurísticas H^1 e H^2 ao contexto de redes de Petri envolveu um estudo das heurísticas da família H^m, em relação a quais aspectos do ambiente de Planejamento Clássico são considerados durante o cálculo, quais as diferenças encontradas entre a linguagem *STRIPS* e as redes de Petri e como a adaptação poderia ser realizada sem que isso acarretasse em perda de informação. As diferenças entre o ambiente de Planejamento Clássico e as redes de Petri foram contornadas através da criação de uma estrutura de dados chamada de *Vetor de Cálculo*, cuja função é, ao mesmo tempo, auxiliar no cálculo da heurística e permitir diversas otimizações para deixar cálculo da heurística mais eficiente.

A presente dissertação é organizada da seguinte maneira. O capítulo 2 apresenta as bases teóricas sobre as quais este trabalho está fundamentado, como os algoritmos clássicos de busca em Inteligência Artificial, os problemas de planejamento clássico, o formalismo das redes de Petri e o algoritmo de desdobramento. O capítulo 3 apresenta a aplicação das heurísticas planejamento H^1 e H^2 no auxílio do desdobramento das redes de Petri. Serão apresentadas as estruturas utilizadas na construção do vetor de cálculo, as modificações da ferramenta *Mole* que se fizeram necessárias para que o algoritmo de desdobramento fosse realizado conforme a lógica do algoritmo de busca A*, bem como a extração do plano ótimo da rede de ocorrências.

O capítulo 4 apresenta os resultados experimentais. As redes de Petri testadas foram obtidas a partir de redes de Petri geradas a partir de problemas de planejamento pelo *Petrigraph*. Como o *Petrigraph* gera uma rede de Petri com um número de lugares e transições menor ou igual às redes geradas pelo *PUP* e em ambos os formalismos garante-se que as redes geradas são equivalentes ao problema de planejamento, não foram realizados testes com as redes de Petri geradas a partir do *PUP*. Em cada domínio, foram feitas comparações de tempo entre o desempenho obtido com as heurísticas H^1 e H^2 em relação ao desempenho do *Mole* sem a utilização de heurísticas, bem como com o desempenho da heurística implementada por *Töws* e do planejador *SatPlan* [15], planejador que converte problemas de planejamento em problemas de satisfabilidade de booleanos. Como o desdobramento de redes de Petri possui complexidade exponencial em relação ao número de lugares da rede de Petri [7], foi estabelecido um limite de tempo de 2500 segundos para a execução individual dos testes. Nos testes em que uma solução foi encontrada em um tempo inferior a 2500 segundos, foi feita uma análise do número de expansões realizadas pela rede de Petri, o trabalho total realizado pelo *Mole*, que está relacionado ao número total de eventos contidos na rede de ocorrências e na lista de prioridades

do *Mole*. Nos problemas em que o limite de tempo foi atingido sem que o plano ótimo fosse encontrado, foi analisada a profundidade atingida pelas heurísticas H^1, H^2 e pela execução do *Mole* sem o uso de heurísticas. Também foi analisada a complexidade do vetor de cálculo ao longo das instâncias dos problemas, em relação ao número médio de dependências presentes nos vetores H^1 e H^2. Outras análises foram feitas envolvendo características inerentes à busca A* adaptada ao contexto de desdobramento de redes de Petri, como é o caso do tempo unitário para a realização de uma expansão da rede de Petri.

Finalmente, no capítulo 5 são apresentadas as conclusões, bem como as perspectivas de trabalhos futuros.

2 *Fundamentação Teórica*

Neste capítulo, serão apresentadas as bases teóricas sobre as quais este trabalho é fundamentado. Em primeiro lugar, serão apresentados algoritmos de busca clássicos de Inteligência Artificial, focando em dois algoritmos de busca cega e um algoritmo de busca heurística. Em seguida, serão apresentados conceitos referentes à definição de problemas de planejamento, focando no planejamento clássico. Em seguida, serão apresentados conceitos referentes às redes de Petri, no que tange ao seu formalismo, representação gráfica, dinâmica de funcionamento e técnicas de análise baseadas no espaço de estados. Na sequência, serão apresentados dois trabalhos relacionados, no caso, o *PUP* e o *PetriGraph*, que se concentram na representação de problemas de planejamento em forma de redes de Petri.

2.1 Algoritmos de Busca

Em alguns dos problemas mais interessantes em inteligência artificial não existe uma maneira eficiente de se encontrar uma solução. Frequentemente, mesmo que a solução possa ser gerada passo a passo, pode-se chegar a estados inválidos, que obrigam a busca a ser refeita em parte ou até mesmo completamente. Em problemas que envolvem atividades escalonadas, por exemplo, as restrições associadas a cada atividade podem tornar algoritmos convencionais completamente ineficientes. Outros problemas ainda possuem a característica de *explosão combinatória*, que ocorre quando o número de estados gerados durante a busca cresce exponenciamente. Os algoritmos de busca foram criados para este tipo de problema [3].

O *espaço de busca* define o conjunto de todos os objetos dentre os quais a busca é realizada [5]. Um objeto no espaço de estados pode ser uma configuração de peças em um jogo de xadrez, uma sequência de localidades em um problema de caminho mínimo, ou qualquer outra situação possível que o problema pode atingir. A relação entre os objetos no espaço de busca é feita através de *operadores*. São exemplos de operadores tanto um movimento legal no xadrez como uma rota que liga uma localidade a outra no problema de caminho mínimo.

Um problema pode ser definido através de quatro componentes [24]:

- O *estado inicial*, que descreve a situação inicial do problema a ser resolvido.

- Uma descrição das possíveis *operadores* que modificam o estado do problema. A formulação mais comum usa a *função de sucessão*. Dado um estado `x`, a função `SUCESSOR-FN(x)` descreve um conjunto de pares <`ação, sucessor`>, em que o primeiro elemento refere-se a uma ação possível de ser executada a partir de `x` e o segundo elemento refere-se ao estado sucessor de `x`.

- O *estado final*, que corresponde ao objetivo da busca. O teste que determina se um determinado estado corresponde à solução pode tanto ser feito através de uma definição explícita de um conjunto de estados objetivos, como através de um conjunto de restrições aplicadas aos estados, de modo que qualquer estado que se enquadre nas restrições é considerado um estado objetivo.

- Uma *função de custo*, que determina o custo de execução de cada ação.

O estado inicial e a função de sucessão determinam implicitamente o espaço de busca do problema. Uma maneira de representar o espaço de busca é através de um grafo, onde os nós representam os estados do problema, os operadores representam os arcos que ligam um estado a outro e a função de custo representa os pesos dos arcos. Uma solução de um problema de busca pode ser tanto um estado como um plano. No primeiro caso, são definidas diversas restrições a um estado, sendo que o estado objetivo precisa necessariamente atender a todas estas restrições. No segundo caso, é fornecido um ou mais estados objetivos, sendo que o algoritmo de busca deve buscar um conjunto ordenado de operadores que transformam o estado inicial no estado final. Uma abordagem mais completa deste último tipo de problema de busca é tratada na seção 2.2.

Os algoritmos de busca são métodos de busca que se baseiam na aplicação dos operadores do espaço de busca nos objetivos, com o intuito de se chegar a um estado objetivo. Alguns algoritmos de busca utilizam-se de métricas, ou heurísticas, para estimar a distância até o estado objetivo, enquanto outros não dispõe desta informação, tendo que recorrer à exaustiva enumeração de objetos no espaço de estados [5].

As seções 2.1.1 e 2.1.2 apresentam, respectivamente, o algoritmo de busca em amplitude e o algoritmo de busca de custo uniforme. Uma abordagem sobre as funções heurísticas é apresentada na seção 2.1.3, sendo sequenciada pelas seções 2.1.4 e 2.1.5, que apresentam, respectivamente, a busca gulosa e a busca A*.

2.1.1 Busca em Amplitude

A busca em amplitude (*breath first search*)[24] é uma estratégia de busca da classe de algoritmos de busca cega, ou seja, não utiliza nenhuma informação que exceda as informações

apresentadas na definição do problema. É caracterizada por expandir primeiro os nós com menor profundidade na árvore de busca.

Trata-se de uma busca completa, pois se existir uma solução em uma profundidade finita da árvore, a busca em amplitude vai encontrá-la, sem o risco de entrar em um laço infinito. A primeira solução encontrada pela busca em amplitude não é necessariamente a solução ótima, pois nos casos em que os pesos dos arcos de cada ação são diferentes, a solução que está em menor profundidade pode não ser necessariamente a solução com menor custo. Entretanto, a busca em amplitude garante a solução ótima quando o peso de todos os arcos da árvore de busca é igual.

Para analisar aspectos como o tempo computacional e a memória necessária para completar a busca, considere um espaço de busca hipotético onde cada estado possui exatamente b sucessores. O nó raiz da árvore de busca gera b nós no nível inicial. Em seguida, cada um destes gera mais b nós, gerando assim um total de b^2 nós no segundo nível. Cada um destes gera mais b nodos, gerando assim b^3 nós no terceiro nível, e assim sucessivamente. Se a solução encontra-se no nível d, no pior caso o algoritmo vai expandir o último nó do nível d, gerando assim $b^{d+1} - b$ nós, visto que o nó objetivo não é expandido. Assim, o número total de nós gerados é mostrado na equação 2.1:

$$1 + b + b^2 + b^3 + ... + b^d + (b^{d+1} - b) = O(b^{d+1}) \tag{2.1}$$

Desta forma, chega-se à complexidade computacional da ordem de $O(b^{d+1})$, em uma busca onde cada estado possui b sucessores e a solução está na profundidade d.

2.1.2 Busca de Custo Uniforme

Como foi explicado na seção 2.1.1, a busca em amplitude garante a solução ótima apenas quando todos os custos de expansão dos nós são iguais. A busca de custo uniforme (*uniform cost search*) [24] foi criada para suprir esta deficiência da busca em amplitude.

A busca de custo uniforme não leva em consideração o número de passos do caminho que liga o estado inicial ao estado atual, como faz a busca em amplitude. A ordem em que os nós são expandidos é feita através da equação 2.2. Nesta fórmula, $g(n)$ representa o custo de transformação do nó inicial até o nó atual na árvore de busca, que é obtido através da soma dos pesos de todos os arcos que ligam os nós intermediários entre o nó atual e o nó inicial. Desta forma, é sempre expandido primeiro o nó que possui o menor custo para ser obtido.

$$f(n) = g(n) \tag{2.2}$$

A busca não é completa, pois corre o risco de entrar em um laço infinito caso encontre arcos com custo igual a zero. Por esta razão, pode-se garantir a completude apenas se o custo de cada passo for maior ou igual a uma constante positiva ε. Esta condição também é suficiente para garantir que esta busca seja ótima, pois o valor do caminho de cada nó expandido só tende a crescer à medida que sejam explorados novos nós durante a busca. A primeira solução que for encontrada terá, necessariamente, um custo menor ou igual a qualquer outra solução encontrada posteriormente.

Uma vez que a busca de custo uniforme é guiada pelos custos de caminhos e não por profundidade, a sua complexidade não pode ser caracterizada em termos de números de nós e profundidades. Ao invés disso, será utilizado o símbolo C^*, que representa o custo da solução ótima, assumindo que cada ação custa pelo menos ε. Desta forma, a complexidade de tempo e espaço do algoritmo no pior caso é $O(b^{1+[C^*/\varepsilon]})$. Esta complexidade pode ser muito maior do que b^{d+1}, especialmente quando uma busca envolve arcos com pesos pequenos que não conduzem a uma solução, juntamente com arcos com pesos maiores e mais próximos de uma solução. Neste caso específico, o algoritmo da busca de custo uniforme vai explorar primeiro os níveis da árvore de busca dos arcos de pesos menores para só então começar a explorar os arcos de maior peso, que são mais próximos da solução.

Nos casos em que o custo dos pesos de todos os arcos são iguais e maiores que uma constante positiva ε, a busca de custo uniforme é idêntica à busca em amplitude, tanto no funcionamento como na complexidade.

2.1.3 Heurísticas

O conhecimento específico que orienta as buscas informadas é chamado de *função heurística*, que consiste em uma estimativa do custo do caminho entre um estado qualquer e o estado final. Os algoritmos de busca heurística tendem a encontrar a solução com mais eficiência do que algoritmos de busca cega, pois a utilização de boas heurísticas durante o processo de busca diminui as expansões de nós desnecessários, diminuindo assim o tempo global da busca.

Uma heurística é admissível quando não superestima o custo de caminho mínimo entre o estado corrente e o estado objetivo do problema. Isto significa que, para qualquer estado alcançável a partir do estado inicial, a estimativa de custo obtida da função heurística vai ser sempre menor ou igual ao custo do caminho mínimo que liga o estado corrente ao estado objetivo o problema.

Outra propriedade importante é a monotonicidade. Sendo n um estado alcançável a partir do estado inicial e n' um estado sucessor de n, uma heurística é monotônica quando o valor retornado pela função heurística sobre qualquer estado n é menor ou igual à soma do custo de transformação do estado n no estado n', com o valor retornado pela função heurística aplicada

ao estado n'.

A notação mais utilizada para heurísticas é a função h(n), onde n representa o estado sobre o qual a heurística é calculada. Uma heurística h_2(n) domina uma outra heurística h_1(n) se, para qualquer estado n, o valor de h_2(n) for sempre maior ou igual ao valor de h_1(n).

A criação de heurísticas admissíveis pode ser feita de várias maneiras. Ao analisar um problema e suas restrições, pode-se relaxar uma ou mais restrições do problema, chegando-se assim a uma versão mais simples, sobre a qual o custo da solução pode ser calculado com menor custo computacional. As heurísticas criadas a partir de versões relaxadas do problema são, por definição, admissíveis, pois são justamente as restrições do problema que aumentam o custo da solução e a complexidade do mesmo. Assim, o valor obtido pela versão relaxada do problema vai ser menor ou igual ao custo de caminho mínimo do problema completo.

Heurísticas admissíveis podem também ser obtidas através da divisão do problema em vários subproblemas, realizando o cálculo da heurística a partir do valor obtido pela soma ou dos valores máximos das heurísticas individuais de cada um deles. Como um subproblema possui menos componentes do que um problema completo e consequentemente menos restrições, o valor obtido em cada subproblema é sempre menor ou igual ao valor global do sistema completo. Esta técnica é especialmente efetiva quando trata-se de problemas cuja solução possui complexidade exponencial. Por exemplo, em um problema que possui a complexidade de O(2^n), é muito mais barato computacionalmente encontrar a solução de *m* subproblemas com complexidade O ($2^{\frac{n}{m}}$).

2.1.4 Busca gulosa

A busca gulosa (*greedy search*) [24] é uma busca que pertence à categoria das estratégias de busca informada, ou seja, buscas que utilizam-se de conhecimentos específicos do problema, heurísticas, para ajudar no processo de busca de uma solução. A ordem em que os nós são expandidos é feita através da equação 2.3. Nesta fórmula, *h(n)* representa o valor heurístico quando aplicado ao estado atual. Assim, é sempre expandido primeiro o nó que, conforme o valor indicado pela função heurística, está mais próximo da solução.

$$f(n) = h(n) \tag{2.3}$$

A busca gulosa não é ótima, pois a estimativa da distância entre o nó atual e o nó objetivo feita pela função heurística raramente é perfeita. Além disso, a busca gulosa é incompleta, pois existe a possibilidade deste algoritmo explorar um caminho com profundidade infinita e nunca testar outras possibilidades.

O tempo de busca, no pior caso, é da ordem de O(b^m), onde *m* representa a profundidade

máxima no espaço de busca. Com uma boa heurística, entretetanto, a profundidade pode ser reduzida substancialmente. A redução vai depender diretamente da complexidade do problema em particular e da qualidade da heurística utilizada.

2.1.5 Busca A*

A busca de custo uniforme (seção 2.1.2) é uma busca completa e ótima sempre que todos os pesos dos nós forem maiores que uma constante positiva ε. Entretanto, por ser um algoritmo de busca cega, tende a ser extremamente ineficiente. Já a busca gulosa (seção 2.1.4), por utilizar-se de conhecimentos específicos do problema, tende a encontrar uma solução em menos tempo do que no caso da busca de custo uniforme, possuindo a desvantagem de não ser uma busca ótima nem completa.

O algoritmo de busca A* [24] combina as vantagens da busca de custo uniforme e da busca gulosa. A ordem em que os nós são expandidos é definida pela equação 2.4. Nesta fórmula, *g(n)* representa o custo de transformação do nó inicial no nó atual e *h(n)* representa a estimativa de custo do nó atual até o nó objetivo, ou seja, a função heurística.

$$f(n) = g(n) + h(n) \tag{2.4}$$

Em um problema de busca, considere $\mathtt{n}_0$ como o estado inicial, $\mathtt{n}_g$ como o estado objetivo e `n` como o estado sobre o qual deseja-se calcular a heurística. A função `g(n)` retorna o custo mínimo de transformação do nó $\mathtt{n}_0$ até o nó `n`. Já a função `h(n)` é a função heurística aplicada a `n`, ou seja, retorna a estimativa da distância entre o nó `n` até o nó objetivo $\mathtt{n}_g$. O valor obtido pela função `f(n)` é uma estimativa do custo de menor solução que passa pelo nó `n`. Uma vez obtido o valor de `f(n)`, o nó é inserido em uma lista ordenada pelo valor respectivo de `f(n)`, sendo expandidos inicialmente os nós em que a função `f(n)` retornou o menor valor.

A busca A* é ótima e completa sempre que a heurística for admissível, ou seja, se a heurística nunca superestimar o custo de `n` até o objetivo para qualquer estado `n`.

Uma das maneiras de qualificar uma heurística é através do *fator de ramificação b**. Considerando um problema particular cujo número total de nodos gerados pela busca A* é igual a *N*, encontrando uma solução na profundidade *d*, o fator de ramificação *b** indica quantos nós, em média, foram expandidos em cada nível da árvore para se chegar ao objetivo. O fator de ramificação *b** é definido na equação 2.5.

$$N + 1 = 1 + b^* + (b^*)^2 + ... + (b^*)^d \tag{2.5}$$

Por exemplo, em um problema onde a busca A* encontra a solução na profundidade 5,

explorando 52 nodos, o fator de ramificação efetivo é 1,92. O fator de ramificação efetivo pode variar em função das várias instâncias de problemas, mas costuma ser constante em problemas muito complexos [24].

2.2 Planejamento

Planejamento é o lado racional da ação [9]. É um processo abstrato e deliberado que seleciona e organiza ações antecipando seus resultados, de forma que se chegue a um objetivo pré-definido. Nem todas as atividades requerem um planejamento explícito antes da ação, tal como as atividades em que a pessoa possui habilidade tal que o plano já está implícito, como aquelas em que as atividades podem ser adaptadas livremente enquanto ocorre a ação.

Uma atividade requer planejamento quando ela prevê situações novas, parâmetros ou objetivos complexos, ou quando utiliza situações menos familiares. Também é necessário quando a adaptação é bastante restrita, como ocorre em ambientes envolvendo altos riscos ou altos custos, atividades conjuntas com outras pessoas, ou atividades sincronizadas com um sistema dinâmico [9].

Como planejamento pode ser complicado e oneroso no que diz respeito a tempo e recursos, costuma-se planejar deliberadamente as atividades apenas quando é estritamente necessário ou quando a relação custo-benefício é vantajosa. Os planos não precisam ser necessariamente os planos ótimos, mas apenas planos bons e factíveis [9].

Décadas de pesquisa em Inteligência Artificial e disciplinas relacionadas mostraram que a capacidade humana de raciocinar é extremamente difícil de ser formalizada [4]. Entretanto, a capacidade de raciocinar sobre ações é um componente essencial do comportamento inteligente, sendo de vital importância no estudo de Inteligência Artificial.

2.2.1 Representação Formal

Como planejamento está relacionado a escolher e organizar ações para mudar o estado de um sistema, requer um formalismo que defina um sistema dinâmico. Um sistema de estados-transições, também chamado de sistema de eventos discretos, é definido pela quádrupla $\sum = (S, A, E, \gamma)$, onde [9]:

- $S = \{s_1, s_2, \ldots, s_n\}$ é um conjunto finito ou recursivamente enumerável de estados.
- $A = \{a_1, a_2, \ldots, a_n\}$ é um conjunto finito ou recursivamente enumerável de ações.
- $E = \{e_1, e_2, \ldots, e_n\}$ é um conjunto finito ou recursivamente enumerável de eventos.

- γ: $S \times A \times E \rightarrow 2^S$ é uma função estado-transição.

Nesta definição, cada estado s_i pode ser definido por uma função 2^P, onde P representa um conjunto finito de predicados. Este modelo de estados é a base da linguagem *STRIPS*, apresentada na seção 2.2.3. Os conjuntos A e E representam os agentes que mudam o estado global do sistema. O primeiro conjunto representa as ações que são realizadas diretamente pelo planejador, enquanto que o segundo conjunto representa os eventos aletórios que podem mudar o estado global do sistema e que não são controláveis pelo planejador.

Um sistema estado-transição pode ser representado por um grafo dirigido cujos nós são os estados em S. Se $s' \in \gamma(s,u)$ onde u é um par (a,e), $a \in A$ e $e \in E$, então o grafo contém um arco de s para s' que é rotulado por u.

Um problema de planejamento pode ser definido pela tripla $\rho = (\sum, s_0, g)$, onde:

- $\sum = (S, A, E, \gamma)$ é a definição de um sistema estado-transição.
- s_0 é o estado inicial do problema.
- $g \subseteq S$ é o conjunto de estados que correspondem a um estado objetivo do sistema.

2.2.2 Planejamento Clássico

A definição clássica de problemas de planejamento assume uma representação de mundo baseada em estados [4]. Esta descrição é um conjunto de sentenças em uma linguagem formal, que assumem valores verdadeiro ou falso e que representam o estado global do sistema. No planejamento clássico, são feitas algumas pré-suposições sobre o sistema [9]:

- *Finito*: O sistema $\sum$ possui um número finito de estados.
- *Completamente observável*: O sistema $\sum$ é completamente observável se for possível ter completo conhecimento sobre qualquer estado possível do sistema.
- *Determinístico*: Para cada ação aplicada a um estado, um único estado seguinte é possível, ou seja, para cada estado $s \in S$ e ação $a \in A$, $|\gamma(s,a)| \leq 1$.
- *Estático*: O estado vai permanecer inalterado até que uma ação seja executada. Desta forma, não existirão eventos aleatórios que possam modificar o estado do sistema, sendo as ações os únicos agentes transformadores possíveis. Consequentemente, o conjunto $E = \emptyset$.

- *Objetivos restritos*: O planejador apenas verifica a pertinência dos estados ao estado objetivo g. Não é feita a definição de objetivos condicionais, tais como estados que devem ser evitados ou restrições na trajetória.
- *Plano sequencial*: A solução é uma sequência de ações ordenada linearmente que modificam o estado inicial até que seja atingido o objetivo.
- *Tempo implícito*: As ações não possuem tempo de duração, ocorrendo instantaneamente.

Sendo um problema de planejamento clássico representado por $\rho = (\sum, s_0, g)$, uma solução é uma sequência de ações $(a_1, a_2, .. a_k) \in A$, possuindo um conjunto de estados correspondentes $(s_1, s_2, ... s_k) \in S$, tal que $s_1 = \gamma(s_0,a_1)$, $s_2 = \gamma(s_1,a_2)$, ..., $s_k = \gamma(s_{k-1},a_k)$ e $s_k \in g$.

2.2.3 Linguagem *STRIPS*

A linguagem *STRIPS* [8] foi desenvolvida para representar problemas de planejamento clássico. É uma representação que utiliza uma notação derivada da lógica de primeira ordem, onde estados são representados por um conjunto de literais que podem assumir o valor verdadeiro ou falso, e as ações são representadas por operadores que mudam os valores destes literais.

O desenvolvimento de um linguagem para planejamento clássico inicia-se com uma linguagem de primeira ordem ξ que possui um número finito de predicados, símbolos constantes e nenhum símbolo funcional. Desta forma, cada símbolo de ξ é uma variável ou uma constante. Os símbolos de ξ são representados por sequências de caracteres alfa-numéricas com pelo menos dois caracteres. Um estado é um conjunto de literais de ξ. Como ξ não possui símbolos funcionais, o conjunto S de todos os estados possíveis é finito.

As ações são os operadores cuja função é modificar o estado global do problema. São representadas pela quádrupla o = {identificador, pre, add, del}, onde:

- *identificador*: É uma expressão sintática única que identifica a ação $a \in A$. Pode ser repesentada na forma *n(x_1, ..., x_k)*, onde *n* representa um símbolo operador, e o conjunto x_1, ..., x_k representa os símbolos variáveis, que aparecem em qualquer lugar de o.
- *pre*: Conjunto de literais pertencentes a ξ que representam as pré-condições da ação. A ação só pode ser executada se todos os literais pertencentes ao conjunto pre forem verdadeiros.
- *add*: Conjunto de literais pertencentes a ξ que representam o efeito positivo da ação. Uma vez que a ação seja executada, o valor lógico de cada um dos literais do estado corrente que pertença ao conjunto add torna-se verdadeiro.

- *del*: Conjunto de literais pertencentes a ξ que representam o efeito negativo da ação. Uma vez que a ação seja executada, o valor lógico de cada um dos literais do estado corrente que pertença ao conjunto `del` torna-se falso.

A linguagem *STRIPS* baseia-se na hipótese de mundo fechado, ou seja, sempre que uma variável não for mencionada, assume-se que o valor desta é negativo.

Por um lado, a linguagem *STRIPS* permite a representação de estados e ações de maneira simples, mas também possui algumas limitações, como o fato de não ser uma linguagem tipada, não suportar igualdade de predicados e não ser uma linguagem lógica. Estas restrições não permitem que alguns tipos de problemas possam ser representados nesta linguagem, o que motivou a extensão da linguagem *STRIPS* em outras versões, como é o caso da linguagem *ADL* (*Action Description Language*), que inclui efeitos condicionais, inclusão de disjunção de objetivos, suporte a variáveis tipadas e a hipótese de mundo aberto, ou seja, em que o valor de uma variável não citada é desconhecido.[21]

2.2.4 Linguagem PDDL

A linguagem *PDDL* (*Planning Domain Definition Language*) foi desenvolvida para a primeira competição de planejadores realizada durante o congresso internacional *AIPS* (*Artificial Intelligence Planning and Scheduling Systems*), em 1998 [18]. O objetivo desta linguagem é expressar o domínio do problema de maneira imparcial, especificando os predicados possíveis, quais ações podem ser executadas e que efeitos estas têm sobre o sistema. Alguns planejadores necessitavam de informações adicionais sobre o problema para encontrar uma solução, tais como ações que precisavam ser executadas para que a solução seja atingida. Como a linguagem PDDL descreve o domínio e o problema de maneira neutra, os diferentes planejadores poderiam ser avaliados utilizando um mesmo critério.

Um *domínio* descreve as características gerais de uma família de problemas, tais como os tipos de objetos possíveis em um problema de um determinado domínio e as ações que podem ser executadas. O algoritmo 2.1 mostra um exemplo de domínio escrito na linguagem *PDDL*, representando o problema do mundo de blocos [10].

Algoritmo 2.1: *Exemplo de domínio na linguagem PDDL*

```
1 (define (domain Blocos)
2     (:requirements :strips :typing)
3     (:types bloco)
4
5     (:predicates (sobre ?x - bloco ?y - bloco)
6         (mesa ?x - bloco)
7         (livre ?x - bloco)
8     )
```

```
9
10      (:action empilha
11          :parameters (?x - bloco ?y - bloco)
12          :precondition (and (mesa ?x) (livre ?x) (livre ?y))
13          :effect
14             (and (not (mesa ?x))
15             (not (livre ?y))
16             (sobre ?x ?y)
17             )
18      )
19
20      (:action desempilha
21          :parameters (?x - bloco ?y - bloco)
22          :precondition (and (sobre ?x ?y) (livre ?x))
23          :effect
24             (and (not (sobre ?x ?y))
25             (livre ?y)
26             (mesa ?x)
27         )
28      )
29  )
```

Tendo em vista que poucos planejadores implementam todas as características da linguagem *PDDL*, esta é dividida em subconjuntos de funcionalidades, chamados de *requirements*, que podem ser observados na linha (2) do algoritmo 2.1. A definição das funcionalidades exigidas no domínio é importante para informar ao planejador se este possui condições de resolver um problema deste domínio. Um planejador que não suporte alguma funcionalidade exigida por um domínio não será capaz sequer de interpretar a sintaxe da descrição do problema. Neste exemplo, a funcionalidade `strips` importa os elementos presentes na linguagem *STRIPS* e a funcionalidade `typing` indica que os objetos possuem tipos. A definição da funcionalidade `typing` permite que seja criado o tipo *blocos* na linha (3) e que os objetos utilizados na definição de predicados e ações também tenham tipos. Existem diversas outras funcionalidades que podem ser inseridas no domínio da linguagem *PDDL*, tais como:

- *disjunctive-preconditions*: problemas com múltiplos objetivos, descritos através do operador "`or`".
- *equality*: operador de igualdade "=".
- *conditional-efects*: efeitos condicionais, através do operador "`when`".
- *existencial-preconditions*: inserção do operador "`exists`" na descrição de objetivos.
- *universal-preconditions*: inserção do operador "`forall`" na descrição de objetivos.
- *subgoal-through-axioms*: inserção de subobjetivos.

A função do domínio é a realização de uma descrição genérica de quais objetos e quais ações podem ser utilizadas na construção de um problema. A descrição específica é realizada

na definição do *problema*, onde também é informado o estado inicial e o estado final. O algoritmo 2.2 [10] mostra um exemplo de problema descrito na linguagem *PDDL*. A linha (2) especifica que este problema é uma instância do domínio `Blocos`, descrito no algoritmo 2.1. Na linha (3), são especificados os objetos que compõem o problema. Sobre esta coleção de objetos, o planejador constrói a lista de predicados, substituindo as variáveis que definem os predicados pelos objetos informados, gerando assim predicados como `(sobre A A)`, `(sobre C D)`, `(mesa A)` e `(livre D)`. Da mesma forma, as ações são construídas substituindo as variáveis das definições genéricas de ações pelos objetos informados pela descrição do problema. O resultado é a coleção completa de predicados e ações possíveis para a instância do problema informada.

Algoritmo 2.2: *Exemplo de Problema na linguagem PDDL*

```
1 (define (problem problema1)
2   (:domain Blocos)
3   (:objects A B C D - bloco)
4   (:init (sobre A D) (livre A) (mesa D) (sobre C B) (livre C) (mesa B))
5   (:goal (AND (sobre A B) (sobre B C) (sobre C D)))
6 )
```

Uma vez que o planejador conheça todos os predicados e ações existentes, a construção do plano pode ser realizada a partir do estado inicial do plano, descrito na linha (4) do Algoritmo 2.2. Um estado é considerado uma solução se todos os predicados descritos no conjunto `goal` estiverem nele contidos.

Algoritmo 2.3: *Exemplo de Plano*

```
1 0:(DESEMPILHA A D)
2 1:(DESEMPILHA C B)
3 2:(EMPILHA C D)
4 3:(EMPILHA B C)
5 4:(EMPILHA A B)
```

O Algoritmo 2.3 mostra um exemplo de plano gerado a partir do domínio `Blocos`, descrito no Algoritmo 2.1 e da descrição do problema descrita no Algoritmo 2.2. Este formato é reconhecido pelo software validador de planos VAL, utilizado na 3^a. Competição Internacional de Planejamento [13]. Este *software* recebe como entrada um domínio e uma descrição do problema, ambos descritos na linguagem *PDDL*, juntamente com um ou mais planos gerados por um planejador, informando quais dos planos informados correspondem a uma solução válida para o problema.

2.2.5 Heurísticas de planejamento

Em [11], Haslum e Geffner desenvolveram uma família de heurísticas voltadas a problemas de Planejamento, denominada H^m, que se baseiam na representação da linguagem *STRIPS*. O desenvolvimento das heurísticas baseou-se na observação do modelo *STRIPS* como um problema de alcançabilidade de grafos. Neste problema, os nodos do grafo representam todos os estados possíveis que o modelo *STRIPS* pode assumir e os arcos que ligam os nodos representam as ações que modificam os estados globais do sistema, sendo que o custo destas deve ser sempre maior do que 0.

A partir da concepção do problema de planejamento em um problema de grafos, foi formulada a heurística H*. A função H*(s), representada na equação 2.6, expressa o custo do caminho mínimo que conecta o estado inicial s_0 ao estado final $s \in g$. Para encontrar o custo do caminho mínimo, a heurística H* faz uma *regressão* do estado $s \in g$. O processo de regressão, representado no item (2) da função H* pelo símbolo $R(s)$, consiste em enumerar todas as duplas <s', a>, onde $a \in A$, em que $\forall\ s' \in S \mid s \cap add(a) \neq \emptyset$, $s \cap del(a) = \emptyset$ e $s' = s - add(a) + pre(a)$ [11]. Para cada um dos estados s' gerados pela regressão de s, é realizada uma chamada recursiva da função H*, cujo resultado é somado ao custo de ser executada a ação $a \in A$ que deu origem à regressão, representado no item (2) pelo símbolo $c(a)$. O item (1) da função H* representa o ponto de parada da função recursiva, que é o ponto em que o estado gerado a partir da recursão coincide com o estado inicial do problema. O resultado da função H* é o valor mínimo obtido dentre todos os estados s', gerados a partir das regressões de s.

$$H^*(s) = \begin{cases} 0, & s \subseteq s_0 \quad (1) \\ \min\limits_{<s',a> \in R(s)} [c(a) + H^*(s')], & s \not\subseteq s_0 \quad (2) \end{cases} \tag{2.6}$$

Apesar da heurística H* parecer atraente à primeira vista, ela possui complexidade exponencial em relação ao número de literais do problema, sendo tão complexa quanto a própria solução do problema. Além disso, quanto maior o número de literais presentes em um estado s, maior a complexidade da realização das regressões de s. As heurísticas da família H^m, propostas por [11], são uma versão relaxada da heurística H*, que ao invés de trabalhar com um estado completo, divide o estado em subconjuntos de tamanho menor ou igual a *m* e em seguida faz a regressão de cada um destes subconjuntos. O resultado da função é o maior valor obtido dentre as regressões dos subconjuntos dos estados de s'. A equação 2.7 representa a fórmula da função H^m.

$$H^m(\alpha) = \begin{cases} 0, & \alpha \subseteq s_0 \quad (1) \\ \min\limits_{<\beta,a>\in R(\alpha)} [c(a) + H^m(\beta)], & |\alpha| <= m,\ \alpha \not\subseteq s_0 \quad (2) \\ \max\limits_{\beta\subset\alpha, |\beta|=m} H^m(\beta), & |\alpha| > m,\ \alpha \not\subseteq s_0 \quad (3) \end{cases} \tag{2.7}$$

Da mesma forma que a heurística H*, a entrada da heurística H^m é o estado final $s \in g$, que é representado na fórmula descrita na figura 2.7 pelo símbolo α. Caso o número de elementos que compõe o estado α seja maior do que o valor de *m*, este estado é dividido em todos os subconjuntos possíveis de tamanho *m*, sendo feita uma nova chamada recursiva da função H^m para cada um deles, conforme é mostrado no item (3). O subconjunto que possuir o maior valor representa o valor heurístico da função. Nos casos em que o número de elementos de α for menor ou igual a *m*, o comportamento da função H^m (figura 2.7) é idêntico à função H* (figura 2.6), pois a função H^m realiza a regressão de α utilizando o mesmo processo que a função H*, tal como mostra o item (2). Além disso, a condição de parada das chamadas recursivas na função H^m também é a mesma da função H*, como mostra o item (1).

Como as heurísticas da família H^m partem da divisão do problema completo em vários subproblemas com menor complexidade, elas são, por definição, admissíveis, representando diferentes graus de complexidade e eficiência [11]. Quanto maior o valor fixado de *m*, maior é a aproximação do valor obtido pela função H^m em relação à função H*, mas também maior é o custo computacional para obtê-las. Para qualquer inteiro positivo *m*, a complexidade de calcular a função H^m é de ordem polinomial N^m, sendo N o número de literais que descreve o problema [11].

Como já foi apresentado nesta seção, a regressão de um conjunto de literais α envolve encontrar uma ação que preenche dois requisitos:

- Deve existir pelo menos uma interseção entre α e o conjunto *add* da ação.
- Não deve existir interseções entre α e o conjunto *del* da ação.

Uma vez que seja encontrada uma ação que atenda estes dois requisitos, o processo de regressão gera outro conjunto de literais, eliminando todos os literais que pertençam ao conjunto *add* da ação e acrescentando a ele todos os elementos do conjunto *pre* da ação.

Ao fixar o valor de *m* da heurística H^m em 1, obtém-se a heurística H^1, cuja fórmula é mostrada na figura 2.8. Esta heurística faz a regressão apenas sobre conjuntos α de tamanho 1. Como α possui um único elemento, é necessário que este elemento esteja presente no conjunto *add* da ação para que o primeiro requisito da regressão seja satisfeito. Na heurística H^1, o segundo requisito da regressão é tautológico, pois toda ação que pertença a um problema de

planejamento coerente não possui interseção entre seus conjuntos *add* e *del*. O resultado final da regressão na função H^1 vai ser exatamente o conjunto *pre* da ação, pois ao eliminar os literais de α que pertençam ao conjunto *add*, obtém-se o conjunto vazio, que somado ao conjunto *pre* resulta no próprio conjunto *pre*. O item (2) da fórmula H^1 ilustra esse processo de regressão, que possui a peculiaridade de ignorar completamente o conjunto *del*.

$$H^1(\alpha) = \begin{cases} 0, & \alpha \subseteq s_0 \quad (1) \\ \min\limits_{\alpha \subseteq add(a)} [c(a) + H^1(pre(a))], & |\alpha| = 1, \ \alpha \not\subseteq s_0 \quad (2) \\ \max\limits_{\beta \subset \alpha, |\beta|=1} H^1(\beta), & |\alpha| > 1, \ \alpha \not\subseteq s_0 \quad (3) \end{cases} \tag{2.8}$$

A função heurística H^2, mostrada na figura 2.9, fixa os subconjuntos α em tamanho menor ou igual a 2. O item (2) da função H^2 demonstra a regressão quando o tamanho de α é igual a 1. Neste caso, o procedimento da regressão é idêntico ao procedimento realizado pela função H^1. Ao aumentar o tamanho dos conjuntos α para 2, situação mostrada no item (4) da função, realizar a regressão passa a ser mais complexo do que simplesmente verificar a pertinência de α ao conjunto *add* das ações, e retornar o conjunto *pre* da ação.

A regressão determina, primeiramente, que deve existir interseção entre o conjunto α e o conjunto *add* da ação. Quando o conjunto α possui tamanho igual a 2, é necessário que pelo menos um dos dois elementos pertença ao conjunto *add*. Se ambos os elementos de α pertencerem ao conjunto *add*, novamente o conjunto *del* vai ser desconsiderado, pois ao eliminar ambos os elementos de α, obtém-se o conjunto vazio, que somado ao conjunto *pre* resulta no próprio conjunto *pre*. Esta operação corresponde à relação `L(p&q)` descrita no item (3.1) da função H^2. Entretanto, quando apenas um dos dois elementos de α pertence ao conjunto *add* da ação, é necessário verificar se o outro elemento pertence ao conjunto *del* da ação, pois a regressão só pode ocorrer se não houver interseção entre α e o conjunto *del* da ação. Uma vez que não exista uma interseção entre os dois conjuntos, a regressão vai eliminar o primeiro elemento e em seguida vai adicionar o conjunto *pre*. Este processo corresponde à relação `L(p|q)`, apresentada nos itens (3.2) e (3.3) da função H^2.

$$H^2(\alpha) = \begin{cases} 0, & \alpha \subseteq s_0 \quad (1) \\ \min\limits_{\alpha \subseteq add(a)} [c(a) + H^2(pre(a)], & |\alpha| = 1, \alpha \not\subseteq s_0 \quad (2) \\ \min\limits_{\alpha=(p,q)} = \begin{cases} \min\limits_{a \in L(p\&q)} [c(a) + H^2(pre(a))] & (3.1) \\ \min\limits_{a \in L(p|q)} [c(a) + H^2(pre(a) \bigcup \{q\})] & (3.2) \\ \min\limits_{a \in L(q|p)} [c(a) + H^2(pre(a) \bigcup \{p\})] & (3.3) \end{cases} & |\alpha| = 2, \alpha \not\subseteq s_0 \quad (3) \\ \max\limits_{\beta \subset \alpha, |\beta|=2} H^2(\beta), & |\alpha| > 2, \alpha \not\subseteq s_0 \quad (4) \end{cases} \tag{2.9}$$

Neste trabalho não serão implementadas heurísticas de ordem maior do que 2, tendo em vista o seu alto custo computacional [11]. A implementação das heurísticas H^1 e H^2 é mostrada no capítulo 3.

2.3 Redes de Petri

As redes de Petri se constituem em uma ferramenta matemática e gráfica de modelagem, desenvolvidas a partir da tese de doutorado *Kommunikation mit Automaten*, de C. A. Petri, em 1962 [22]. Este trabalho atraiu a atenção de diversos pesquisadores, por possuir mecanismos de análise poderosos, que permitem a verificação de propriedades e da corretude do sistema especificado [17]. Uma rede de Petri pode modelar sistemas paralelos, concorrentes, assíncronos e não-determinísticos, se adaptando bem a aplicações nas quais noções de eventos e de evoluções simultâneas são importantes.

As vantagens da utilização da Rede de Petri podem ser resumidas nas seguintes considerações [23]:

- Pode-se descrever uma ordem parcial entre vários eventos, deixando claro quais eventos são *dependentes* entre si, quais são independentes e quais são concorrentes.
- Os estados e os eventos do sistema são representados explicitamente.
- Os sistemas podem ser representados em diferentes níveis de abstração sem mudar a descrição da linguagem.
- A representação da rede é feita através de uma descrição precisa e formal, o que torna possível a verificação de propriedades do sistema.

2.3.1 Elementos básicos

Uma rede de Petri é um grafo bipartido, direcionado e ponderado, formado por três elementos básicos [2]:

- *Lugar*: Pode ser interpretado como uma condição, um estado parcial, ou um conjunto de recursos. É representado por um círculo e geralmente é associado a um predicado, como "*máquina livre*" ou "*peça em espera*".
- *Transição*: É interpretada como um evento que modifica o estado global do sistema, como por exemplo "*iniciar a operação*". É representada por uma barra ou um retângulo.
- *Marcação*: É um indicador binário associado a cada lugar da rede, que representa uma condição booleana.
- *Arcos*: Os arcos são os elementos que ligam apenas lugares a transições ou transições a lugares.

A dinâmica de funcionamento de uma rede de Petri é apresentada na seção 2.3.3.

2.3.2 Representação Formal

Formalmente, uma rede de Petri elementar pode ser definida pela tripla $N = <P, T, F>$, onde:

- P é um conjunto finito de lugares, de dimensão *n*.
- T é um conjunto finito de transições, de dimensão *m*, onde $P \cap T = \emptyset$.
- $F \subseteq (P \times T) \cap (T \times P)$ é uma relação binária, sendo a relação de fluxo de N, indicando os arcos que ligam transições a lugares e lugares a transições.

Sendo R uma rede de Petri, então:

- ${}^\bullet x = \{y \mid y\ F_n\ x\}$, y é o pré-conjunto de x.
- $x^\bullet = \{y \mid x\ F_n\ y\}$, y é o pós-conjunto de x.

Uma rede de Petri Condição/Evento é uma quádrupla $N = \{P, T, F, M_0\}$, onde:

- (P, T, F) constituem uma rede de Petri elementar.
- $M_0 \subseteq P$ constitui o estado inicial da rede.

2.3.3 Dinâmica de Funcionamento

Em uma rede de Petri, o estado global é definido pela distribuição de marcas nos lugares que a compõe, sendo cada lugar considerado um subestado do sistema. A figura 2.1 apresenta um exemplo de rede de Petri. O estado desta rede é definido pelo conjunto {p2, p4}, visto que são estes os lugares que possuem uma marca.

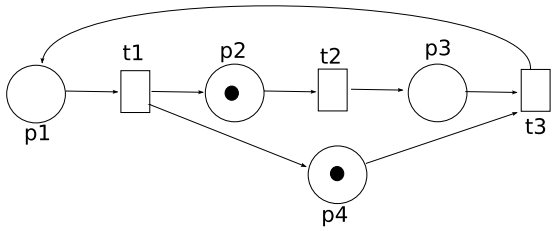


Figura 2.1: Exemplo de rede de Petri

A mudança de estado do sistema ocorre através do disparo das transições. Um disparo ocorre se e somente se todos os lugares precedentes da transição tenham marcas de número igual ou maior do que o peso do arco. No caso da rede da figura 2.1, a única transição disparável é transição *t2*, visto que o pré-conjunto de *t2* é composto pelo lugar *p2*, que possui uma marca.

O disparo de uma transição é feito em duas fases. Em primeiro lugar, as marcas dos lugares precedentes da transição são removidos. Em seguida, são depositadas novas marcas nos lugares de incidência posterior das transições. A figura 2.2 ilusta a nova configuração da rede mostrada na figura 2.1 após o disparo da transição *t2*. Nesta nova configuração, a marca presente no lugar *p2* é consumida, sendo inserida uma nova marca no lugar *p3*.

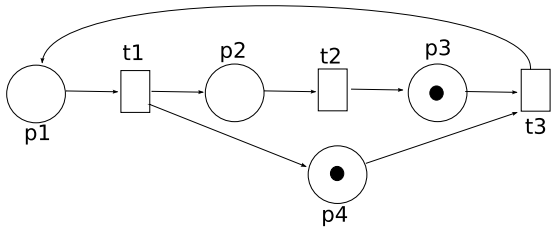


Figura 2.2: Exemplo de rede de Petri após o disparo

O novo estado da rede é definido pelo conjunto {p3, p4}. Uma vez que a transição *t2* tenha sido disparada, a nova configuração habilita a transição *t3* para o disparo.

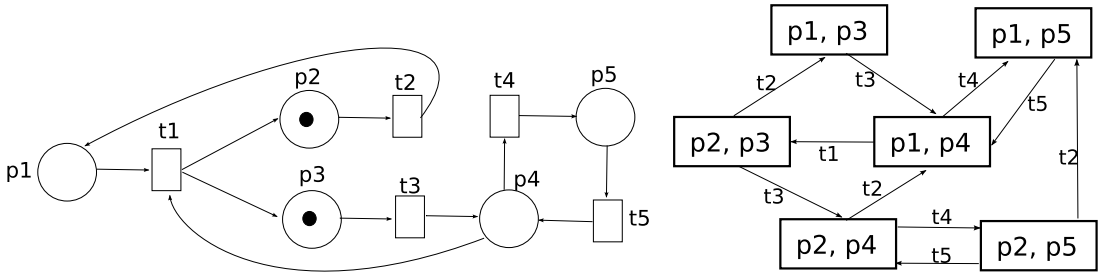
2.3.4 Alcançabilidade em redes de Petri

Sendo uma rede Condição/Evento definida por $N = \{P, T, F, M_0\}$, o problema de alcançabilidade consiste em verificar se uma dada marcação M_g é alcançável a partir da marcação inicial M_0. Para a marcação ser alcançável, deve existir uma execução entrelaçada M_0 $[t_1> M_1$ $[t_2> M_2$, ..., M_{n-2} $[t_{n-1}> M_{n-1}$ $[t_n> M_n$, de tal modo que $M_g \subseteq M_n$.

A verificação de alcançabilidade de uma rede de Petri pode considerar tanto buscar uma marcação completa, onde $M_g = M_n$, como uma submarcação, que equivale aos casos onde $M_g \subset M_n$. O problema de alcançabilidade de submarcação é teoricamente equivalente ao problema de alcançabilidade de uma marcação completa. Em redes de Petri acíclicas, o problema de alcançabilidade é NP-Completo e em redes k-limitadas o problema de alcançabilidade é PSPACE-Completo.

Uma das maneiras de solucionar problemas de alcançabilidade de redes de Petri é através da utilização do *grafo de alcançabilidade*. O grafo de alcançabilidade é um grafo direcionado gerado a partir de uma rede de Petri, onde os nós representam cada estado possível da rede e os arcos representam as transições disparáveis a partir de cada estado. A vantagem da utilização do grafo de alcançabilidade é a possiblidade da utilização de algoritmos de busca clássicos e técnicas heurísticas para encontrar a marcação objetivo a partir da marcação inicial. Entretanto, a explosão combinatória do número de estados possíveis torna viável a utilização de grafos de alcançabilidade apenas em redes pequenas.

A figura 2.3(a) apresenta uma rede de Petri Condição/Evento e a figura 2.3(b) apresenta o grafo de alcançabilidade relativo a esta rede.



(a) Rede de Petri (b) Grafo de alcançabilidade

Figura 2.3: Exemplo de construção de grafo de alcançabilidade

Uma outra maneira de representar o espaço de estados, utilizando uma estrutura mais compacta do que o grafo de alcançabilidade, é a técnica de *desdobramento*, que será tratada na seção 2.4.

2.4 Desdobramento

A técnica de desdobramento, do inglês *unfolding*, foi desenvolvida por McMillan e Probst em [19] com o objetivo de reduzir o problema da explosão combinatória de estados encontrado no grafo de alcançabilidade de redes de Petri. A partir de uma rede de Petri cíclica, o processo de desdobramento gera uma rede de Petri acíclica, denominada *rede de ocorrências*, que contém todas as marcações possíveis da rede original.

A seção 2.4.1 apresenta as restrições presentes na rede de ocorrências, que é gerada pelo processo de desdobramento. Os eventos de corte e as configurações, que definem o ponto de parada do processo de desdobramento, são apresentados na seção 2.4.3. A seção 2.4.2 apresenta o processo de ramificação, que é o a base do algoritmo de desdobramento. A seção 2.4.4 apresenta uma ferramenta voltada ao desdobramento de redes de Petri.

2.4.1 Rede de Ocorrência

Sendo $N = \{P, T, F\}$ uma rede de Petri elementar, N é chamada de rede de ocorrência se e somente se:

(i) $\forall p \in P, |{}^\bullet p| \leq 1$.

(ii) $\forall t \in T, |{}^\bullet t| \geq 1$ e $|t^\bullet| \geq 1$.

(iii) O fechamento transitivo de $F_k(F_k^+)$ é irreflexivo, ou seja, $x_1 F_1 x_2 F_2$, ..., $F_k x_n$, $x_1 \neq x_n$.

O item (i) garante que, em uma rede de ocorrência, todas as condições possuem, no máximo, uma transição de entrada. O item (ii) implica que todas as transições possuem conjuntos pré e pós diferentes de vazio. O item (iii) demonstra que a rede de ocorrência deve ser acíclica, pois um fechamento transitivo irreflexivo implica que não existe um caminho que liga qualquer elemento da rede a ele mesmo.

2.4.2 Processo de Ramificação

Seja uma rede de ocorrências definida por $ON = \{B, E, F\}$, onde:

- B é um conjunto de condições.
- E é um conjunto de eventos.
- $F\ F \subseteq (B \times E) \cap (E \times B)$ é a relação de fluxo de ON.

Um processo de ramificação de uma rede de Petri consiste em construir uma rede de ocorrências associada a um sistema de rotulamento, que preserva todas as informações relativas a conflitos e concorrência da rede original. A ramificação de uma rede de Petri $N = \{P, T, F, M_0\}$ consiste na dupla $\beta = \{ON, \varphi\}$, onde ON equivale à rede de ocorrência $ON = \{B, E, F\}$ e φ equivale à função de rotulamento. A função de rotulamento φ deve satisfazer as seguintes condições:

- $\varphi(B) \subseteq P$ e $\varphi(E) \subseteq T$ (condições são mapeadas para lugares, e eventos para transições).
- $\forall\ e \in E$, $\varphi\{|{}^\bullet e|\} = {}^\bullet\varphi(e)$ e $\varphi\{|e^\bullet|\} = \varphi(e)^\bullet$ (preserva o ambiente das transições).
- Sendo Min o conjunto de elementos de $B \cap E$ que possuem o seu pré-conjunto vazio, $\varphi\{|Min(ON)|\} = M_0$ (o conjunto de condições sem pré-condição corresponde à marcação inicial).
- $\forall e_1, e_2 \in E$, se ${}^\bullet e_1 = {}^\bullet e_2$ e $\varphi(e_1) = \varphi(e_2)$ então $e_1 = e_2$ (Não existe redundância nas transições).

O processo de ramificação pode ser repetido infinitamene. Entretanto, é possível truncá-lo em uma subrede, denominada prefixo finito completo, que contém todas as marcações alcançáveis a partir da rede de entrada N. A construção do prefixo completo da rede utiliza a noção de configurações da rede e de eventos de cortes, que serão expostos na seção 2.4.3.

2.4.3 Configuração e Cortes

Sendo uma rede de ocorrência $ON = \{B, E, F\}$, uma configuração C representa um conjunto de eventos que satisfazem as seguintes condições [19]:

- $\forall\ e_1, e_2 \in E \mid e_2 \preceq e_1, e_1 \in C \Rightarrow e_2 \in C$.
- $\forall\ e_1, e_2 \in C \mid e_1 \neq e_2 \Rightarrow \bullet e_1 \cap \bullet e_2 = \emptyset$

Para um conjunto de eventos ser considerado uma configuração, é necessário que, para qualquer evento do conjunto, estejam inseridas todos os eventos precedentes e que não exista conflitos entre nenhum par de eventos. A figura 2.4 mostra o prefixo completo da rede de Petri da figura 2.3. O conjunto de eventos {e1, e2, e3, e5} não pode ser considerado uma configuração, pois existe uma interseção entre o pré-conjunto dos eventos e3 e e5. Da mesma forma, o conjunto de eventos {e3, e4} não pode ser considerado uma configuração por não conter o evento e2, precedente a e3.

A configuração local de um evento é a configuração mínima na qual um evento pode estar inserido. A configuração local do evento e4 é representada por [e4] = {e2, e3, e4}. O conjunto de eventos {e1, e2, e3, e4} constitui uma configuração válida, mas o evento e1 não pertence à configuração local de e4.

Um evento e é um evento de corte (*cut-off*) se a marcação da rede após o disparo do evento for exatamente igual à marcação da rede após o disparo de qualquer evento pertencente à sua configuração local, ou à marcação inicial da rede. O evento e5 é um evento de corte, pois a marcação associada ao evento e5 é {p2, p3}, idêntica à marcação inicial da rede.

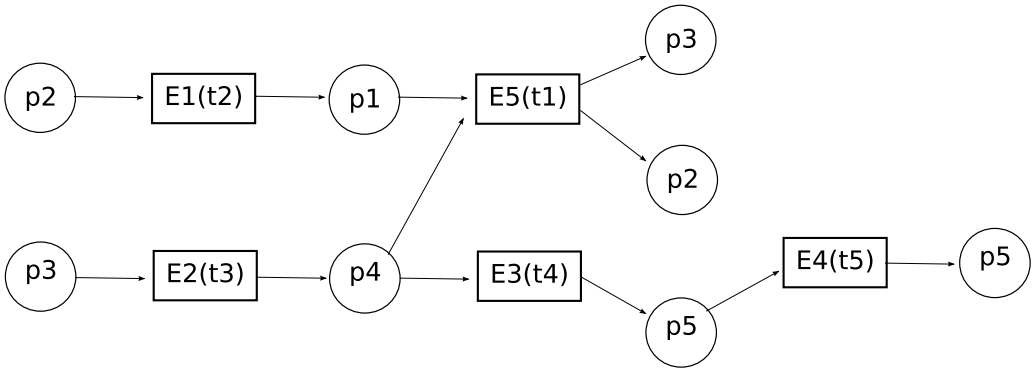


Figura 2.4: Prefixo completo de uma rede de Petri

O algoritmo *ERV Unfolding*, apresentado no algoritmo 2.4, foi criado por Esparza, Römer e Vogler [7] para gerar o prefixo completo de uma rede de Petri. É a base da ferramenta de desdobramento de redes de Petri *Mole* [25], apresentada na seção 2.4.4

Algoritmo 2.4: *ERV Unfolding*

Input: Um $\mathcal{SN}$ = {N,M_0}, sendo que M_0 = {p1 , . . . , pk}
Output: Unfolding Unf de $\mathcal{SN}$
`Unf` ← lugares de M_0;
`pe` ← transições habilitadas por M_0;
`cut-off` ← Ø;
while *pe* ≠ Ø **do**
  escolha um evento `e` = (t, X) de `pe` tal que [`e`] seja mínimo;
  if *([e] ∩ cut-off)* = Ø **then**
    adicione `e` e seu pós-conjunto em `Unf`;
    `pe` ← **PE**(`Unf`);
    if *(`e` é um evento de corte)* **then**
      `cut-off` ← `cut-off` ∪ `e`;
    end
  else
    `pe` ← `pe` \ {`e`}
  end
end

Em primeiro lugar, o algoritmo *ERV Unfolding* recebe como entrada uma rede Condição/Evento, a partir da qual o desdobramento será construído. A rede de ocorrências `Unf` é iniciada com os lugares pertencentes à marcação inicial da rede; a lista de eventos a serem expandidos `pe` é inicializada com as transições habilitadas a partir da marcação inicial e a lista de eventos de corte `cut-off` é inicializada com vazio.

Uma vez iniciado o algoritmo de desdobramento, o evento com menor configuração local é retirado da lista de eventos a serem expandidos. Se o evento não for um evento de corte, então ele e seu pós-conjunto são adicinados à rede de ocorrência, processo que se chama *expansão da*

rede de ocorrências. Em seguida, a partir de cada transição disparável sobre o estado resultante do disparo da transição pertencente ao evento `e`, é gerado um novo evento que é adicionado à lista de eventos a serem expandidos.

A condição de parada ocorre quando todas as marcações alcançáveis a partir de M_0 já tiverem sido representadas na rede de ocorrências `Unf`. Neste ponto, qualquer evento que esteja presente na lista `pe` será considerado um evento de corte. Em consequência, não será gerada uma nova expansão a partir dele.

O tamanho da rede de ocorrências gerada a partir de uma rede de Petri cíclica é exponencialmente maior do que a rede de Petri original, mas costuma ser exponencialmente menor do que o espaço de estados que ela representa [7].

2.4.4 Mole

O *Mole* [25] é uma ferramenta que realiza o desdobramento de redes de Petri seguras, disponibilizado sob Licença Pública Geral (GNU GPL). Foi desenvolvido por Stefan Römer e Stefan Schwoon a partir do algoritmo *ERV Unfolding*, apresentado na seção 2.4.3, sendo mantido pelo grupo de Sistemas Paralelos do Departamento de Ciência da Computação da Universidade de Oldenburg na Alemanha.

Esta ferramenta foi projetada para ser compatível com o ambiente do projeto PEP (*Programming Environment based on Petri Nets*) [28], da própria Universidade de Ondenburg. A entrada do programa é um arquivo com a descrição da rede no formato próprio do ambiente e a saída é um arquivo com o desdobramento da rede. O formato da saída é compatível com a ferramenta *graphviz* [6], que pode convertê-lo para um formato gráfico.

O algoritmo 2.5 mostra um exemplo de entrada para o *Mole*, referente à rede de Petri descrita na figura 2.3.

Algoritmo 2.5: *Exemplo de arquivo de entrada do Mole*

```
1  PEPc
2  PTNet
3  FORMAT_N
4  PL
5  1"p1"0@0M0m0
6  2"p2"0@0M1m1
7  3"p3"0@0M1m1
8  4"p4"0@0M0m0
9  5"p5"0@0M0m0
10 TR
11 1"t1"0@0
12 2"t2"0@0
13 3"t3"0@0
14 4"t4"0@0
15 5"t5"0@0
```

```
16 6"t6"0@0
17 TP
18 1<2
19 1<3
20 2<1
21 3<4
22 4<5
23 5<4
24 PT
25 1>1
26 2>2
27 3>3
28 4>1
29 4>4
30 5>5
```

A partir do arquivo de entrada, o *Mole* realiza o desdobramento utilizando o mesmo algoritmo desenvolvido por [7], descrito na seção 2.4.3. A marcação inicial da rede é inserida na rede de ocorrências e as transições disparáveis a partir da marcação inicial são inseridas na lista de prioridades que possui a função de armazenar os eventos a serem expandidos. Em seguida, são realizadas sucessivas expansões da rede de ocorrências até o momento em que todas as marcações alcançáveis estejam presentes na rede de ocorrências.

Um dos parâmetros de entrada da ferramenta *Mole* é a transição de parada. Se a transição de parada for especificada, o algoritmo de desdobramento vai ser realizado até o momento em que ocorra a expansão da rede a partir de um evento que contenha esta transição. Uma vez que isso aconteça, o processo de desdobramento é interrompido automaticamente.

2.5 Trabalhos relacionados

Nesta seção, serão apresentados os planejadores *PUP* [12] e *Petrigraph* [29]. O funcionamento destes planejadores é feito em duas fases: na primeira fase, problemas de planejamento descritos em linguagem *PDDL* são traduzidos em forma de uma rede de Petri, a fim de que o problema de planejamento seja convertido em um problema de alcançabilidade. Na segunda fase, as redes de Petri geradas são submetidas a um processo de desdobramento, de maneira que o plano possa ser obtido a partir da rede de ocorrências.

O planejador *Petrigraph* é um aperfeiçoamento do planejador *PUP*, no que se refere à construção das redes de Petri a partir dos problemas na liguagem *PDDL*. Na segunda fase do *Petrigraph*, o desdobramento da rede de Petri foi feito com o auxílio de uma heurística derivada da heurística H^1, apresentada na seção 2.2.5, modificando a ferramenta *Mole* para realizar a busca com base na lógica do algoritmo de busca gulosa, apresentado na seção 2.1.4.

A presente dissertação é focada na segunda parte do planejador *Petrigraph*, modificando a ferramenta *Mole* para realizar a busca com base na lógica do algoritmo de busca A*, apresentado

na seção 2.1.5, utilizando-se das heurísticas de planejamento H^1 e H^2, apresentadas na seção 2.2.5. Os experimentos apresentados no capítulo 4 são feitos tomando como base as redes de Petri geradas pelo *Petrigraph*, além de ser feita uma comparação de tempo entre a heurística implementada por Töws em [29] e as heurísticas da família H^m.

2.5.1 PUP - *Planning via Unfondings of Petri Nets*

O *PUP* é um planejador desenvolvido por Hickmott e colegas em [12], que traduz problemas de planejamento descritos na linguagem *PDDL* em redes de Petri. Esta tradução é feita com o intuito de converter o problema de planejamento em um problema de alcançabilidade de redes de Petri. O objetivo é aproveitar-se de algumas vantagens provenientes do formalismo inerente às redes de Petri, como o fato destas apresentarem uma descrição compacta do espaço de busca, além de representar concorrência e relações causais entre ações de uma maneira que torna possível a identificação de subproblemas.

Para o desenvolvimento do *PUP*, foram estudadas técnicas de análise de redes de Petri que pudessem ser aplicadas com sucesso pela análise dos problemas de planejamento. A técnica de desdobramento, apresentada na seção 2.4, mostrou-se atrativa por ser uma técnica de análise de alcançabilidade que preserva e explora com menor custo computacional a estrutura da rede de Petri inerente. Além disso, o processo de desdobramento gera uma rede mais simples, chamada de rede de ocorrências, que é acíclica e mantém apenas conflitos de primeira ordem (apenas entre dois elementos). Em relação à complexidade, a rede de ocorrências possui tamanho exponencial em relação ao tamanho da rede de Petri de origem, mas pode ser exponencialmente menor do que a representação do espaço de busca através do grafo de alcançabilidade, apresentado na seção 2.3.4. Além disso, o plano obtido através do processo de desdobramento pode ser extraído da rede de ocorrências em tempo linear em relação ao seu tamanho.

A solução do problema de planejamento através do planejador *PUP* ocorre em duas fases. Na primeira fase, o problema de planejamento é traduzido para uma rede de Petri segura. Na segunda fase, é realizado o desdobramento da rede de Petri com o auxílio de uma heurística, de modo que o plano possa ser extraído da rede de ocorrências resultante.

A primeira fase da execução do planejador *PUP* é o mapeamento do problema de planejamento em um problema equivalente no qual todas as ações são seguras. Este processo é necessário porque a rede de Petri resultante precisa ser segura para possibilitar o uso da ferramenta de desdobramento *Mole*, que só aceita redes de Petri deste tipo. Entretanto, conforme [12], o uso de redes de Petri seguras também auxilia na representação dos operadores proposicionais de planejamento, que se baseiam em variáveis lógicas. No caso de uma rede de Petri k-limitada, seria necessário a utilização de uma estrutura não trivial para garantir a equivalência entre a rede de Petri e o problema de planejamento, visto que a inserção de várias marcas em

um mesmo lugar manteria seu valor como verdadeiro, ao passo que seria necessário remover todas as marcas do lugar para tornar o seu valor falso.

Sendo $o = \{p, e\}$ uma ação de um problema de planejamento, onde p representa as pré-condições da ação (conjunto *pre*) e e representa seus efeitos, ou melhor, a união dos efeitos positivos (conjunto *add*) e dos efeitos negativos (conjunto *del*) da ação. O mapeamento do problema de planejamento em um problema equivalente com ações seguras é feito através da substituição da ação o por $2^{|e'|}$ ações distintas, onde e' representa todos os literais que estão presentes em *e* e não estão presentes em p. Por exemplo, a ação $o = \{p, e\}$ onde $p = \{a, \neg b, c\}$ e $e = \{\neg a, b, d, \neg e\}$, é descrita da seguinte maneira:

$o = \{p,e\}$	$p = \{a, \neg b, c\}$	$e = \{\neg a, b, d, \neg e\}$	
$o_1 = \{p_1, e_1\}$	$p_1 = \{a, \neg b, c, d, \neg e\}$	$e_1 = \{\neg a, b\}$	$e'_1 = \{\}$
$o_2 = \{p_2, e_2\}$	$p_2 = \{a, \neg b, c, \neg d, \neg e\}$	$e_2 = \{\neg a, b, d\}$	$e'_2 = \{d\}$
$o_3 = \{p_3, e_3\}$	$p_3 = \{a, \neg b, c, d, e\}$	$e_3 = \{\neg a, b, \neg e\}$	$e'_3 = \{\neg e\}$
$o_4 = \{p_4, e_4\}$	$p_4 = \{a, \neg b, c, \neg d, e\}$	$e_4 = \{\neg a, b, d, \neg e\}$	$e'_4 = \{d, \neg e\}$

Nas ações resultantes, pode-se encontrar ainda pré-condições negativas. A fim de possibilitar o mapeamento do problema em redes de Petri, é necessário a eliminação das pré-condições negativas, visto que as redes de Petri só utilizam pré-condições positivas. A eliminação das pré-condições negativas ocorre substituindo-se cada variável $\neg v$ por uma variável $\hat{v}$, que têm seu valor oposto a v. Deste modo, as ações resultantes da eliminação das pré-condições negativas são mostradas a seguir:

$o_1 = \{p_1, e_1\}$	$p_1 = \{a, \hat{b}, c, d, \hat{e}\}$	$e_1 = \{\hat{a}, \neg a, \neg\hat{b}, b\}$
$o_2 = \{p_2, e_2\}$	$p_2 = \{a, \hat{b}, c, \hat{d}, \hat{e}\}$	$e_2 = \{\hat{a}, \neg a, \neg\hat{b}, b, \neg\hat{d}, d\}$
$o_3 = \{p_3, e_3\}$	$p_3 = \{a, \hat{b}, c, d, e\}$	$e_3 = \{\hat{a}, \neg a, \neg\hat{b}, b, \neg e, \hat{e}\}$
$o_4 = \{p_4, e_4\}$	$p_4 = \{a, \hat{b}, c, \hat{d}, e\}$	$e_4 = \{\hat{a}, \neg a, \neg\hat{b}, b, \neg\hat{d}, d, \neg e, \hat{e}\}$

A partir do conjunto de ações geradas pela eliminação das pré-condições negativas, o problema de planejamento pode ser mapeado para uma rede de Petri. Seja $N = \{P, T, F\}$ a definição de uma rede de Petri. Os lugares da rede de Petri são formados pelo conjunto de variáveis das ações, ou seja, $P = A \cup \hat{A}$. As transições da rede de Petri são formadas pelo conjunto de ações o_i resultantes, ou seja, $T = O$. A relação de fluxo F é obtida através de $\iota = \{p, e\} \in T$, de tal forma que $\{(a, \iota) \mid a \in p\} \cup \{(\iota, a) \mid (a \in e) \vee (a \in e) \wedge (\neg a \neg \in e)\}$.

A rede de Petri resultante é equivalente ao problema de planejamento, sendo que os lugares de rede mapeiam as variáveis de estado do problema de planejamento e as transições mapeiam as ações que modificam o estado global. A próxima fase é a solução do problema de

alcançabilidade da rede criada através do algoritmo de desdobramento. A definição do estado objetivo do problema é feita a partir de uma transição inserida na rede de Petri, denominada *GOAL*. A pré-condição da transição *GOAL* corresponde ao estado objetivo do problema, tal como definido pelo problema de planejamento.

A ferramenta *Mole* é configurada para realizar o desdobramento da rede de Petri gerada até o momento em que a transição *GOAL* for disparada. Uma vez que o desdobramento tenha sido encerrado, o plano ordenado do problema é obtido a partir da configuração local do evento que possui a transição *GOAL*. Como a configuração local de um evento corresponde à menor configuração à qual o evento pode fazer parte, este plano contém apenas as ações necessárias para a transformação do estado inicial da rede no estado objetivo.

A lista de prioridades da versão original do *Mole*, que contém os eventos ainda não expandidos, é ordenada pelo tamanho da configuração local de cada evento. Isto significa que o desdobramento é realizado através da lógica do algoritmo de busca em amplitude, apresentado na seção 2.1.1. Como já foi discutido, o algoritmo de busca em amplitude não é eficiente no que se refere ao uso de memória e de tempo. Neste caso específico, o desdobramento é realizado apenas com o objetivo de obter uma rede de ocorrências que contenha o plano, e não o prefixo completo da rede.

Com o intuito de otimizar a busca realizada pelo algoritmo de desdobramento, forma incorporadas à ferramenta *Mole* heurísticas de planejamento para guiar o processo de desdobramento da rede de Petri, modificando a ordenação da lista de prioridades de forma que o desdobramento se comporte conforme a lógica do algoritmo A*, apresentado na seção 2.1.5. O algoritmo de busca A* ordena os nós abertos do grafo de acordo com a função `f(x) = g(x) + h(x)`, onde `g(x)` equivale ao custo de obtenção do nó `x` e a função `h(x)` equivale a uma função heurística que estima o custo do nó `x` até o estado objetivo.

Sendo `C(e)` a configuração local de um evento `e`, a função `g(e)` é definida como $g(C) = \sum_{e \in C} cost(\varphi(e))$. Como a configuração local é a menor configuração à qual um evento pode fazer parte, a função `g(C)` retorna o custo mínimo exato de obtenção do evento `e`.

A função `h(e)` é obtida pela aplicação de uma heurística à marcação resultante do evento `e`. Nos testes realizados pelo *PUP*, foram utilizadas as seguintes heurísticas. O símbolo $\prec_h$ indica que a ordenação de eventos a serem expandidos é feita através da somatória do tamanho da configuração local com a heurística.

- $\prec_0$: Busca de custo uniforme, ou seja, *h(s) = 0*
- $\prec_{h^1}$: Heurística H^1, ou seja, $h(s) = h^1(s, G)$
- $\prec_{h^1_+}$: Heurística H^1_+, ou seja, $h(s) = h^1_+(s, G)$

A heurística $\prec_0$ indica que a ordenação de eventos a serem expandidos é feita apenas pelo tamanho da configuração local dos eventos, pois o valor heurístico aplicado a todos os lugares é igual a 0. Esta heurística corresponde à execução natural do *Mole*, que já utiliza a configuração local dos eventos para ordenar a fila de eventos a serem expandidos. A heurística $\prec_{h^1}$ equivale à heurística H^1, representada na equação 2.10. Esta heurística faz parte da família de heurísticas H^m, apresentada na seção 2.2.5.

$$H^1(\alpha) = \begin{cases} 0, & \alpha \subseteq s_0 \quad (1) \\ \min\limits_{\alpha \subseteq add(a)} [c(a) + H^1(pre(a))], & |\alpha| = 1, \ \alpha \not\subseteq s_0 \quad (2) \\ \max\limits_{\beta \subset \alpha, |\beta|=1} H^1(\beta), & |\alpha| > 1, \ \alpha \not\subseteq s_0 \quad (3) \end{cases} \tag{2.10}$$

A heurística $\prec_{h^1_+}$, apresentada na equação 2.11, é obtida substituindo-se, na linha (3), a operação *max* pela operação de somatória. Isto significa que, ao invés de retornar o valor máximo entre as regressões dos subconjuntos, a função H^1_+ soma o valor obtido a partir da regressão dos subconjuntos. A heurística resultante não é admissível, mas é mais informativa do que a heurística H^1.

$$H^1_+(\alpha) = \begin{cases} 0, & \alpha \subseteq s_0 \quad (1) \\ \min\limits_{\alpha \subseteq add(a)} [c(a) + H^1_+(pre(a))], & |\alpha| = 1, \ \alpha \not\subseteq s_0 \quad (2) \\ \sum\limits_{\beta \subset \alpha, |\beta|=1} H^1_+(\beta), & |\alpha| > 1, \ \alpha \not\subseteq s_0 \quad (3) \end{cases} \tag{2.11}$$

A versão modificada do *Mole* que acrescenta o uso das heurísticas para guiar o processo de desdobramento pelo *PUP* não é disponibilizada, o que impossibilita a verificação da mesma.

2.5.2 *Petrigraph* - Um algoritmo para planejamento em redes de Petri

O *Petrigraph* foi desenvolvido por Töws em [29] como um aperfeiçoamento do método de mapeamento de problemas de planejamento em redes de Petri realizado pelo PUP, descrito na seção 2.5.1. A desvantagem do método de mapeamento realizado pelo *PUP* é que o processo de criação da rede de Condição/Evento não é otimizado. A rede de Petri resultante possui exatamente o dobro de variáveis em relação ao número de variáveis de estados do problema de planejamento. Uma vez que a complexidade do prefixo finito completo da rede de Petri, no pior caso, é da ordem de $O(2^n)$, onde n é o tamanho da rede, uma diminuição do número de condições da rede de Petri mapeada do problema de planejamento terá um efeito exponencial no tamanho da rede de ocorrências, e consequentemente no tempo de execução do algoritmo de

desdobramento.

Com o intuito de reduzir o número de lugares e de transições na rede de Petri resultante, o *Petrigraph* utilizou-se da noção de variáveis multivaloradas e de grafo de transição de domínios. A geração da rede de Petri pelo *Petrigraph* é definida nos seguintes passos:

1. Remover do problema de planejamento os predicados constantes.
2. Remover do problema de planejamento os predicados de caminho único.
3. Análise dos predicados que podem ser expressos em forma de variáveis multivaloradas.
4. Construir a representação de grafos de transição de domínio a partir das variáveis de estados multivaloradas.
5. Mesclar as redes de Petri obtidas no passo 4.

Os predicados constantes são aqueles que não são removidos por nenhuma ação de domínio. Por esta razão, eles não interferem no desdobramento da rede de Petri, não precisando serem representados por variáveis de estados. Já os predicados de caminho único são aqueles que são removidos por ações do domínio, mas que não são adicionados por nenhuma ação. Estes predicados somente precisam ser representados por variáveis de estados se fizerem parte da solução do problema, conforme a descrição do estado objetivo. Os predicados constantes e de caminho único podem ser encontrados através de uma busca nos conjuntos *pre* e *add* nas ações do domínio. A complexidade da busca destes predicados na descrição do problema possui complexidade da ordem de *O(n)* em relação ao número de ações.

As variáveis multivaloradas foram introduzidas por Jonsson e Bäckström em [14] como uma proposta para reduzir o número de variáveis de estado necessárias para a descrição de um problema de planejamento. As variáveis multivaloradas constituem um grupo de variáveis de estado mutuamente exclusivas, ou seja, em qualquer estado do problema alcançável a partir do estado inicial, apenas uma das variáveis deste grupo terá valor verdadeiro e todas as outras terão valor falso.

O processo de busca dos grupos de variáveis multivaloradas inicia-se com a definição de um grupo G, onde inicialmente são inseridas as variáveis de estado presentes na descrição do estado inicial do problema. Em seguida, para cada item em G, se houver um operador no domínio que tenha G como pré-condição, todas as pós-condições deste operador são adicionads a *G*. Este processo é repetido até que nenhuma variável de estado nova seja adicionada a *G*.

O conjunto G possui a função de agrupar todas as variáveis de estado que influenciam diretamente a busca da solução do problema. Uma vez que este conjunto esteja formado, procede-se à análise dos predicados do domíno PDDL para verificar se estes estão balanceados, ou caso

contrário, efetuar procedimentos a fim de torná-los balanceados. Um predicado `pred` com *n* argumentos é balanceado em relação a $i \leq n$ se toda ação que remover um predicado *pred*(p^1, ..., p^i, ..., p^n) adiciona um, e apenas um, predicado *pred'*(p^1, ..., p^{j}', ..., p^n).

Um predicado *pred* pode não ser balanceado, mas este pode ser combinado com outros predicados que tornem o grupo balanceado, caso sejam encontradas ações complementares, onde sempre uma ação remove o predicado *pred* adicionando *pred'* e outra ação que remove *pred'* adicionando *pred*. O processo recursivo de balanceamento dos predicados ocorre da seguinte maneira:

- Verifica-se se o predicado *pred* é balanceado em relação a *i*. Caso afirmativo, o algoritmo retorna o conjunto ⟨*pred*,*i*⟩.
- Se predicado não for balanceado, verifica-se todas as ações que removam o predicado *pred* com argumentos (p1 ... pn), e uma busca é feita nas pós-condições destas ações por:
  - pred'(p1, ..., p_j', ..., p_{i-1}, p_{i+1}, ..., p_n)
  - pred'(p1, ..., p_{i-1}, p_{i+1}, ..., p_j', ..., p_n)
  - pred'(p1, ..., p_{i-1}, p_{i+1}, ..., p_n)
- Para cada predicado encontrado desta maneira, pode-se efetuar a combinação de predicados da seguinte forma: Um predicado *pred+pred'* é criado com argumentos (p_1, ..., p_{i-1}, p_i", p_{i+1}, ..., p_n) e todas as ocorrências de *pred* e *pred'* no domínio são substituídas pelo novo predicado.
- Caso *pred'* tenha um argumento p_i', os valores possíveis para o argumento p_i" são tanto os de p_i como os de p_i', sem que haja qualquer interseção entre os dois. Senão, p_i" pode assumir todos os valores de p_i , mais um novo valor $v_{pred'}$, que será utilizado nos casos em que *pred+pred'* substitui *pred'*.
- Para cada predicado criado desta maneira, substitui-se o predicado criado no domínio e o processo é refeito.

O resultado deste algoritmo é um conjunto de predicados balanceados possíveis no problema. Cada predicado balanceado em relação a *i* é representado por uma única variável de estado multivalorada, com valores entre 0 e *m*, onde *m* representa o número de objetos que podem ser definidos como o argumento p_i do predicado.

Em seguida, as condições que mudam o valor das variáveis de estados multivaloradas são descritas a partir de um grafo de transição de domínios, ou DTG. Cada nó do DTG se torna um lugar na rede de Petri, os arcos se tornam transições e uma marca é inserida no lugar gerado a partir de um predicado que esteja presente no estado inicial do problema.

Todas as redes derivadas de DTGs são combinadas em uma única rede de Petri. Se existirem duas ou mais transições com o mesmo nome, estas são combinadas, mantendo todos os arcos de entrada e saída. A rede continua segura, pois o número de arcos de entrada de cada transição sempre é igual ao número de arcos de saída. Se duas redes derivadas de DTGs diferentes possuírem lugares provindos da mesma variável, estes também podem ser combinados.

Para as variáveis de estados não pertencentes a DTGs, o mesmo algoritmo de conversão utilizado pelo *PUP* é aplicado, utilizando-se de variáveis complementares que garantam que a rede seja segura, eliminando-se as pré-condições negativas, traduzindo o modelo para o formato de rede de Petri e mesclando a nova rede com a rede gerada a partir das DTGs.

Assim como ocorre no planejador *PUP*, em [29] o mapeamento do problema de planejamento em forma de uma rede de Petri é realizado com o intuito de converter um problema de planejamento em um problema de alcançabilidade de redes de Petri, de modo que a solução seja encontrada com o auxílio do algoritmo de desdobramento de redes de Petri. O *Mole* é executado até o momento em que seja alcançada a transição objetivo, no caso, a transição *GOAL*, cujo disparo encerra o desdobramento para que o plano ordenado possa ser extraído da rede de ocorrências. A proposta de [29] é a incorporação da heurística H^1, apresentada na seção 2.2.5, ao algoritmo de desdobramento, de modo a guiar o processo de desdobramento até encontrar a solução. Como o trabalho de Töws pertence ao mesmo grupo de pesquisa ao qual este trabalho faz parte, foi possível ter acesso ao código-fonte desta versão modificada do *Mole* e realizar a análise da implementação da heurística H^1. Esta análise mostrou diferenças substanciais entre a heurística implementada e a heurística proposta por Haslum e Geffner em [11].

Enquanto a heurística H^1 se baseia em regressões das condições da rede de Petri, partindo do estado objetivo até o estado corrente, a heurística implementada por Töws na ferramenta *Mole* é feita a partir da rotulagem dos lugares e das transições da rede de Petri, inserindo números inteiros que indicam a quantos níveis de distância os elementos estão da transição *GOAL*.

O algoritmo 2.6 apresenta o trecho do código-fonte extraído desta versão modificada do *Mole*, cuja função é realizar a rotulagem da rede de Petri. O procedimento *CostTransition* é executado assim que a ferramenta *Mole* termina a leitura da rede de Petri a partir do arquivo de entrada. A primeira chamada deste procedimento associa a transição *GOAL* com o custo igual a 1.

Como o valor de rotulagem dos elementos da rede, tanto de lugares como de transições, é iniciado com 0, a presença do valor 0 em qualquer um destes elementos indica que ele ainda não foi rotulado. Por esta razão, a linha (4) verifica se o valor do rótulo da transição de entrada é igual a 0 ou maior do que o custo de entrada da chamada do procedimento *CostTransition*. Caso afirmativo, o custo da transição é atualizado com o novo valor na linha (5). Uma vez que o custo da transição tenha sido atualizado, a linha (6) começa a rotular os lugares da rede de

Petri presentes no conjunto de pré-condições da transição. Da mesma forma como aconteceu com as transições, os lugares só são rotulados se ainda não tiverem sido rotulados, ou se o valor do rótulo for maior do que o novo valor, conforme mostra a linha (8). Em seguida, o rótulo dos lugares são atualizados com o mesmo custo da transição na linha (9) e o procedimento prepara-se para a realização de uma chamada recursiva. Nesta chamada, o símbolo `p->preset` está relacionado com o conjunto de todas as transições às quais o lugar representado pelo símbolo `p` está presente no conjunto de pós-condições, ou seja, $\mathtt{p} \subseteq \mathtt{t}^{\bullet}$. Como esta transição está um nível abaixo da transição de entrada do procedimento *CostTransition*, o valor de custo é acrescido de 1.

Algoritmo 2.6: *Procedimento de rotulagem da rede*

```
1  void CostTransition(trans_t *t, int cost) {
2          place_t *p;
3          nodelist_t *n, *m;
4          if (t->value==0 || t->value > cost)     {
5                  t->value=cost;
6                  for (n=t->preset; n; n=n->next) {
7                          p=n->node;
8                          if (p->value==0 || p->value > cost)     {
9                                  p->value=cost;
10                                 for (m=p->preset; m; m=m->next)
11                                         CostTransition(m->node, cost+1);
12                         }
13                 }
14         }
15 }
```

Sendo $\mathtt{N} = \{\mathtt{P}, \mathtt{T}, \mathtt{F}, \mathtt{M}_0\}$ uma rede Condição/Evento e $\mathtt{M}_e$ a marcação associada ao evento `e`, criado durante o desdobramento da rede de Petri, o valor heurístico atribuído ao evento `e` é definido pela fórmula da equação 2.12, onde $\mathtt{p} \in \mathtt{P}$ e `p->value` é o rótulo atribuído ao lugar `p`:

$$H^{pg} = \sum_{p->value} p \in M_e \tag{2.12}$$

Existem algumas diferenças substanciais entre a heurística implementada nesta versão modificada do *Mole* e a heurística H^1 de Haslum e Geffner. A heurística H^1 se baseia em regressões do estado objetivo do sistema. Parte-se do estado objetivo, sendo realizadas sucessivas regressões até alcançar o estado corrente, que é o estado sobre o qual a heurística é calculada. As variáveis de estado pertencentes ao estado corrente são inicializadas com zero, e as outras variáveis recebem um valor que indica o menor número de regressões necessárias para que esta variável se torne verdadeira, partindo do estado corrente. No caso da heurística de Töws, os lugares do estado objetivo são inicializadas com valor igual a 1, sendo o valor dos outros lugares são acrescidos levando em conta a relação de fluxo binária entre lugares e transições da rede de Petri.

A função H^{pg} se limita a somar o valor dos rótulos dos lugares marcados do estado corrente. A fila de eventos a serem expandidos é ordenada de acordo com o valor da soma dos rótulos, sendo expandidos primeiro os eventos com menor valor. O tamanho da configuração local dos eventos não é levada em consideração. Deve-se considerar também que esta função não é admissível, o que implica que, mesmo que o tamanho da configuração local seja incorporado à heurística H^{pg} para que o desdobramento se comporte como o algoritmo A*, a busca não garante o plano ótimo.

3 Aplicação das Heurísticas H^1 e H^2 no contexto de redes de Petri

Este trabalho foi desenvolvido em continuidade de pesquisas envolvendo planejamento e redes de Petri, realizadas pelo grupo de pesquisas LIAMF (Laboratório de Inteligência Artificial e Métodos Formais), da Universidade Federal do Paraná. Em [26], é apresentado o *Petriplan*, uma abordagem em que o grafo de planos é transformado em uma rede de Petri acíclica e o problema de alcançabilidade é resolvido através de técnicas de programação inteira, envolvendo a equação fundamental das redes de Petri. Em seguida, foi proposta a Rede de Planos [27], onde foi abandonado o grafo de planos para ser utilizada uma estrutura mais eficiente para a representação do problema de planejamento em uma rede de Petri acíclica.

Outros trabalhos do grupo começaram a transcrever os problemas de planejamento em uma rede de Petri cíclica. Em [20], foram estudadas regras de transformação para conversão dos problemas de Planejamento em redes de Petri seguras e em [29], foi proposto o *Petrigraph*, apresentado na seção 2.5.2, que utiliza o conceito de variáveis multivaloradas para a otimização do algoritmo *PUP* [12], apresentado na seção 2.5.1.

Tendo em vista a proximidade entre modelos de Planejamento Clássico e das redes de Petri, neste trabalho buscou-se a adaptação das heurísticas de planejamento da família H^m para guiar o processo de desdobramento de redes de Petri até uma marcação específica. Estas heurísticas de planejamento foram propostas por Haslum e Geffner em [11] e são apresentadas na seção 2.2.5. Com o intuito de realizar esta adaptação, foi feito um estudo minuncioso das heurísticas da família H^m, bem como do algoritmo de desdobramento de redes de Petri proposto por Esparza, Römer e Vogler em [7] e das estruturas de dados utilizadas pela ferramenta *Mole*.

Neste capítulo, serão apresentadas as estruturas de dados utilizadas que permitiram a aplicação das heurísticas H^1 e H^2 no contexto das redes de Petri, bem como as modificações feitas na ferramenta *Mole* para que a busca ocorra de acordo com o algoritmo A*. A seção 3.1 descreve as estruturas de dados utilizadas para a realização da adaptação de uma heurística oriunda do ambiente de planejamento clássico no contexto de redes de Petri. A seção 3.2 mostra as modificações necessárias na ferramenta *Mole*, bem como o cálculo da heurística a partir do vetor de cálculo e as otimizações necessárias para o cálculo mais eficiente da heurística. A seção

3.3 mostra um exemplo detalhado de cálculo, onde pode-se visualizar as estruturas de dados e a dinâmica do algoritmo A* acoplado à ferramenta *Mole*. Finalmente, a seção 3.4 apresenta considerações referentes às estruturas de dados utilizadas.

3.1 Vetor de Cálculo da heurística

Um problema de Planejamento Clássico baseado em *STRIPS* possui muitas semelhanças com o formalismo das redes de Petri. Ambos os formalismos definem os estados a partir de um conjunto de predicados binários e são baseados em ações que modificam o estado global.

A adaptação das heurísticas de planejamento ao contexto de redes de Petri foi feita a partir de uma estrutura de dados que contém as regressões de todos os subconjuntos de lugares da rede de Petri, que neste trabalho será chamada de "*vetor de cálculo*". O armazenamento dos subconjuntos em uma estrutura permite que todo o processamento destinado à construção de regressões dos conjuntos seja feito como pré-processamento, como também o cálculo da heurística com auxílio de um algoritmo não-recursivo e algumas otimizações no cálculo da heurística.

A seção 3.1.1 apresenta a transcrição das ações para o formato *STRIPS*, processo que é realizado para contornar as diferenças entre as ações do ambiente de planejamento e as transições das redes de Petri. A seção 3.1.2 apresenta a construção propriamente dita do vetor de cálculo.

3.1.1 Transcrição das transições em forma *STRIPS*

As heurísticas H^1 e H^2 [11] foram projetadas para o ambiente de planejamento clássico baseado em *STRIPS*. Na linguagem *STRIPS*, a maneira como as ações afetam o estado global é descrita através de três conjuntos de variáveis de estado, no caso os conjuntos *add*, *pre* e *del*, conforme mostrado na seção 2.2.3. As regressões do estado global, que constituem-se na essência do funcionamento das heurísticas da família H^m, ocorrem de acordo com a pertinência das variáveis de estado aos conjuntos que definem as ações.

No caso das redes de Petri, a própria dinâmica de funcionamento do formalismo deixa implícito como a transição influencia o estado global da rede, pois no disparo de uma transição, todas as marcas das pré-condições da transição ($^\bullet$t) são consumidas e todas as pós-condições da transição ($t^\bullet$) recebem uma marca.

A transcrição das transições em forma *STRIPS* possui o objetivo de representar a informação de como a transição influencia o estado global da mesma maneira como as ações da linguagem *STRIPS* fazem. Esta informação, que antes era implícita ao formalismo de redes de Petri, passa a ser expressa de maneira explícita, em três conjuntos: *add*, *pre* e *del*. Com esta representação, é possível realizar a regressão de subconjuntos de lugares da rede de Petri, tal como ocorre na

linguagem *STRIPS*.

Nas transições transcritas, o conjunto *pre* corresponde exatamente ao conjunto de pré-condições da transição de origem ($^\bullet$t), visto que o conjunto *pre* informa as condições necessárias para o disparo desta. Os conjuntos *add* e *del* equivalem aos conjuntos de pós-condições e pré-condições da transição, respectivamente, eliminando-se de ambos os laços elementares. A eliminação é importante porque a presença de um laço elementar envolvendo um lugar e uma transição em uma rede de Petri indica que este lugar é necessário para o disparo da transição, mas o disparo não influencia o seu estado, visto que a marca é recolocada no lugar após ser consumida. Por definição, o conjunto *add* indica os lugares não-marcados que passam a ter uma marca após o disparo da transição, e o conjunto *del* indica os lugares marcados que tem a sua marca eliminada após o disparo da transição.

Formalmente, a transcrição das transições em forma *STRIPS* é feito da seguinte maneira: Sendo $t \in T$ o conjunto de transições da rede de Petri e $a_t \in A$ o conjunto de transições transcritas, $\forall\ t \in T$:

- $pre(a_t) = {}^\bullet t$
- $add(a_t) = t^\bullet \setminus ({}^\bullet t \cap t^\bullet)$
- $del(a_t) = {}^\bullet t \setminus ({}^\bullet t \cap t^\bullet)$

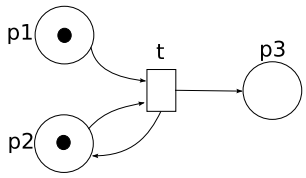


Figura 3.1: Exemplo de transição

A figura 3.1 mostra uma transição representada ao lado dos lugares a ela diretamente relacionados. O conjunto de pré-condições desta transição é composto pelos lugares *p1* e *p2* e o conjunto das pós-condições corresponde aos lugares *p2* e *p3*. A transcrição desta transição em forma *STRIPS* deste sistema é formada pelos seguintes conjuntos:

- **PRE**: p1, p2
- **ADD**: p3
- **DEL**: p1

As transições transcritas em forma *STRIPS* apresentam a mesma informação que as mostradas nas redes de Petri. A partir deste conjunto de transições transcritas, é construído o vetor de cálculo, conforme apresentado na seção 3.1.2.

3.1.2 Construção do vetor de cálculo

O cálculo das heurísticas da família H^m, apresentado na seção 2.2.5, é realizado através de um algoritmo recursivo que divide o conjunto de literais em vários subconjuntos de tamanho *m*, sendo feitas regressões sucessivas desses subconjuntos até se chegar ao estado inicial s_0. A estrutura dos problemas a que estas heurísticas se aplicam é estática, pois não são criados ou excluídos predicados e/ou ações durante o processo de busca. Isso implica que o valor heurístico atribuído a um subconjunto de predicados sempre vai depender das mesmas ações geradoras. Por esta razão, durante o cálculo do algoritmo recursivo da heurística para uma determinada marcação, independentemente de quantas vezes um mesmo subconjunto de predicados seja calculado, o valor heurístico encontrado para este subconjunto será sempre o mesmo. Além disso, partindo do pressuposto que o número de predicados do problema é finito e enumerável, então o número de subconjuntos de tamanho menor ou igual a *m* também é finito e enumerável.

Neste trabalho, a enumeração de subconjuntos de lugares foi realizada através da criação de uma estrutura de dados chamada *vetor de cálculo*, cuja função é enumerar todos os conjuntos de predicados de tamanho menor ou igual a *m*, bem como todas as regressões possíveis a partir de cada um deles.

A figura 3.1 mostra a fórmula básica das heurísticas da família H^m. A linha (2) indica que a heurística é calculada extraindo-se o custo mínimo encontrado dentre todas as regressões possíveis em um conjunto de predicados α. O conjunto β representa a regressão de α a partir da ação *a*. Nos casos em que o número de elementos de α é maior do que *m*, o conjunto α é "quebrado" em quantos subconjuntos β de tamanho *m* forem necessários, conforme indica o comando da linha (3).

$$H^m(\alpha) = \begin{cases} 0, & \alpha \subseteq s_0 \quad (1) \\ \min\limits_{<\beta,a>\in R(\alpha)} [c(a) + H^m(\beta)], & |\alpha| <= m, \ \alpha \not\subseteq s_0 \quad (2) \\ \max\limits_{\beta\subset\alpha, |\beta|=m} H^m(\beta), & |\alpha| > m, \ \alpha \not\subseteq s_0 \quad (3) \end{cases} \tag{3.1}$$

A função do vetor de cálculo é enumerar todos os subconjuntos de lugares da redes de Petri de tamanho menor ou igual a *m*, de acordo com a ordem da heurística trabalhada. Como todos os subconjuntos possíveis estão presentes no vetor de cálculo, pode-se garantir que, qualquer que seja a regressão do conjunto α produzida no comando da linha (2) da fórmula da figura 3.1, todos os subconjuntos de tamanho menor ou igual a *m* produzidos pelas regressões também estarão presentes no vetor de cálculo.

Conforme apresentado na seção 2.2.5, a regressão de um determinado conjunto de literais $s \in S$ é feita a partir de uma ação $a \in A$, onde exista interseção entre *s* e o conjunto *add* de *a* e não

exista interseção entre *s* e o conjunto *del* de *a*. Se este requisito é verdadeiro, então a regressão é feita excluindo-se do conjunto *s* todos os elementos que pertençam ao conjunto *add* de *a* e acrescentando-se a *s* todos os elementos que pertençam ao conjunto *pre* de *a*. No planejamento clássico, a regressão de literais é realizada a partir do conjunto de ações *STRIPS* presentes no modelo formal da definição do problema. No caso deste trabalho, a regressão dos conjuntos de lugares é realizada a partir das transições transcritas pelo processo apresentado na seção 3.1.1.

O vetor de cálculo enumera todas as regressões dos subconjuntos de tamanho menor ou igual a *m* com o auxílio de uma outra estrutura, que será chamada de *conjunto de dependências*. O conjunto de dependências é construído através dos processos das linhas (2) e (3) da fórmula da figura 3.1. Para cada subconjunto α presente no vetor de cálculo, são geradas todas as regressões β que são viáveis a partir do conjunto de transições transcritas. Uma vez que as regressões de α tenham sido geradas, cada um dos conjuntos β são divididos em subconjuntos de tamanho menor ou igual a *m*. Como todos os subconjuntos β possuem uma referência ao vetor de cálculo, o conjunto de dependências de α é construído armazenando-se a posição que estes subconjuntos β possuem no vetor de cálculo. O resultado final do processo de geração dos conjuntos de dependências é, para cada conjunto α presente no vetor de cálculo, um conjunto de dependências $c_{a_t} \in C_\alpha$, onde cada elemento c_{a_t} é gerado a partir da regressão de α através da transição transcrita a_t.

A diferença entre a implementação das heurísticas H^1 e H^2 é a construção do vetor de cálculo de heurística, tanto no que se refere ao número de elementos como na construção das regressões. Conforme [11], a partir da heurística H^3, a explosão combinatória de elementos torna o cálculo da heurística muito caro computacionalmente, tanto no que se refere a tempo de execução como em memória utilizada. Por esta razão, neste trabalho não será discutida a implementação de heurísticas de ordem maior do que 2.

O vetor de cálculo H^1 trabalha com conjuntos de tamanho 1, tendo o mesmo número de elementos que o número de condições da rede de Petri. Na heurística H^1, a regressão de α é equivalente ao conjunto *pre* da transição que deu origem à regressão. Sendo uma rede de Petri representada por $N = <P, T, F>$ e o conjunto de transições transcritas de T representado por $a_t \in A$, o vetor de cálculo H^1 é composto pelos elementos $z_p \in Z$, onde $p \in P$. Cada elemento z_p está associado a um conjunto de dependências C_z, onde $\forall\ a \in A \mid p \in add(a) \Rightarrow c_z \supseteq pre(a)$.

O vetor H^2 é formado por conjuntos de tamanho 2, pois o objetivo da heurística H^2 não é determinar o custo de obtenção de uma única condição, como faz a heurística H^1, mas o custo de se obter as duas condições simultaneamente. A utilização do conjunto *del* aumenta o poder de precisão da heurística, pois a regressão de um par de lugares não é realizada por transições que eliminam um elemento para adicionar o outro.

A construção do vetor H^2 para uma rede de Petri com *n* elementos é feita listando todos

os pares de condições (p, q), onde $p \leq q$. As posições do vetor H^2 onde $p = q$ equivalem às chamadas recursivas da função recursiva H^2 para conjuntos de tamanho 1. Os demais pares sempre são construídos com o segundo par maior que o primeiro para evitar repetição de pares ao longo do vetor H^2. Sendo uma rede de Petri representada por N = <P, T, F> e o conjunto de transições transcritas de T representado por $a_t \in A$, o vetor de cálculo H^2 é formado por todos os conjuntos $z_{p,q} \in Z$, onde $p,q \in P$ e $p \leq q$. Cada elemento de $z_{p,q}$ está associado a um conjunto de dependências c_z, onde:

- $\forall a \in A \mid p \in add(a), q \in add(a) \Rightarrow c_z \supseteq (pre(a))$;
- $\forall a \in A \mid p \in add(a), q \notin add(a), q \notin del(a) \Rightarrow c_z \supseteq (pre(a) \cup q)$;
- $\forall a \in A \mid q \in add(a), p \notin add(a), p \notin del(a) \Rightarrow c_z \supseteq (pre(a) \cup p)$;

Uma maneira encontrada para otimizar o cálculo das heurísticas é armazenar, em uma estrutura separada, o conjunto *pre* das transições transcritas sobre as quais o vetor de cálculo é construído. Desta forma, uma vez que o valor máximo é extraído dentre as dependências presentes nele, este pode ser reaproveitado por qualquer outro elemento do vetor de cálculo que possua uma regressão feita a partir da mesma transição.

Procurar no vetor de cálculo H^2 a indexação de todas as regressões que forem geradas é um processo caro computacionalmente, no que diz respeito a tempo. Por esta razão, foram desenvolvidas fórmulas baseadas na estrutura criada para o vetor H^2. A seção 3.1.3 apresenta a fórmula que retorna o tamanho do vetor H^2 para uma rede de *n* elementos e a fórmula de indexação do vetor H^2, que retorna a posição, no vetor de cálculo H^2, de qualquer elemento $z_{p,q}$.

3.1.3 Fórmulas de indexação do vetor de cálculo H^2

O número de elementos do vetor H^2 é da ordem de $n^2/2$ em relação ao número de condições da rede de Petri. Durante a construção do vetor de cálculo, são criados muitos pares de condições, sendo necessário conhecer a indexação destes pares no vetor de cálculo. Realizar uma busca cada vez que for necessário conhecer a indexação dos pares quando o vetor de cálculo é gerado, ou quando for necessário calcular a heurística sobre uma determinada marcação, demandaria muito tempo. Porém, como o conjunto de condições é finito e estático e está organizado de maneira lógica, é possível conhecer antecipadamente tanto o tamanho de um vetor de cálculo H^2 com *n* elementos, quanto a indexação de qualquer par de elementos (p,q) através de uma fórmula baseada na progressão aritmética, como será demonstrado a seguir.

Seja uma rede de Petri N = <P, T, F> formada por *n* condições e o vetor de cálculo H^2 Z composto pelos elementos $z_{p,q} \in Z$. Neste vetor, existirão exatamente *n* elementos onde $p = 0$,

tais como $z_{0,0}$, $z_{0,1}$, ... , $z_{0,n-2}$, $z_{0,n-1}$. O elemento posterior a $z_{0,n-1}$ é o elemento $z_{1,1}$ e não o elemento $z_{1,0}$, pois o conjunto (1, 0) é idêntico ao conjunto (0, 1), já listado anteriormente. Por esta razão, o vetor H^2 contém exatamente n-1 elementos onde p = 1. Da mesma forma, existirão exatamente n-2 elementos onde p = 2, pois o vetor de cálculo não vai repetir os pares (0, 2) e (1, 2). Desta forma, conclui-se que o número de elementos do vetor H^2 iniciados por um elemento p diminui na mesma proporção em que o valor de p aumenta. O último elemento do vetor de cálculo H^2 é o elemento $z_{n-1,n-1}$. Assim, o número total de elementos do vetor de cálculo H^2 é igual a *n* elementos do tipo $z_{0,n-1}$, *n-1* elementos do tipo $z_{1,n}$, *n-2* elementos do tipo $z_{2,n}$, e assim sucessivamente.

Com base na lógica de construção do vetor H^2, pode-se calcular o número de elementos deste vetor através da fórmula da soma de termos de uma progressão aritmética, onde $a_1 = n$, $a_n = 1$, e $r = -1$. Ao fazer a substituição dos termos, obtém-se a fórmula apresentada na figura 3.2, que representa o tamanho do vetor H^2 para uma rede de Petri de *n* condições:

$$S_n = \frac{n * (1 + n)}{2} \tag{3.2}$$

A lógica de construção do vetor de cálculo H^2 também pode ser utilizada para a construção de uma fórmula que encontre qualquer par de elementos *(p,q)*, eliminando a necessidade de realizar uma busca que retorne a indexação deste par no vetor de cálculo H^2.

Como já foi apresentado, um elemento $z_{p,q}$ do vetor de cálculo H^2 sempre vai ser precedido por n elementos do tipo $z_{0,q}$, (n - 1) elementos do tipo $z_{1,q}$, e assim sucessivamente, até ser precedido por n - p - 1 elementos do tipo $z_{p-1,q}$. Portanto, o número de elementos situados entre $z_{0,0}$ e $z_{p-1,n-1}$ pode ser representado pela somatória dos termos de uma progressão aritmética, onde $a_1 = n$, $a_n = (n - p - 1)$, e $r = (-1)$. A esta fórmula, deve-se acrescentar o número de elementos situados entre $z_{p,p}$ até $z_{p,q}$, que é igual à diferença entre p e q. O resultado é a fórmula apresentada na figura 3.3.

$$index(p,q)_n = \frac{p * (2n - p + 1)}{2} + (q - p) \tag{3.3}$$

A função *index(i, j)* sempre vai retornar uma posição válida do vetor de cálculo H^2 de uma rede de Petri de tamanho *n*, desde que $p \leq q \leq n$.

3.2 Aplicação da heurística no algoritmo de busca

Nesta seção, será discutida a aplicação das heurísticas H^1 e H^2 na solução de problemas de alcançabilidade de redes de Petri utilizando o algoritmo de desdobramento, apresentado na seção 2.4. Foi utilizada a ferramenta de desdobramento *Mole*, que foi modificada para executar

o desdobramento de acordo com as regras estabelecidas pelo algoritmo de busca A*, utilizando como auxílio na busca as heurísticas de busca H^1 e H^2.

O cálculo das heurísticas sobre as marcações dos eventos gerados pelo *Mole* durante o desdobramento é realizado a partir do vetor de cálculo, cuja estrutura é mostrada na seção 3.1. A maneira como o vetor de cálculo é usado para a realização do cálculo das heurísticas é mostrado na seção 3.2.1.

A seção 3.2.2 apresenta a identificação de subconjuntos inalcançáveis, cuja função é otimizar o cálculo da heurística posteriormente. As modificações necessárias na ferramenta *Mole* que possibilitaram a execução do algoritmo de busca A* são apresentadas na seção 3.2.3.

Uma vez que a busca A* é encerrada, o plano ótimo pode ser extraído diretamente da rede de ocorrências gerada pelo processo de desdobramento, processo descrito na seção 3.2.4

3.2.1 Cálculo da Heurística

Nesta seção, será apresentado o algoritmo utilizado para o cálculo das heurísticas H^1 e H^2 sobre uma determinada marcação, utilizando como auxílio o vetor de cálculo, cuja estrutura é descrita na seção 3.1.2.

O cálculo da heurística é feito pelo preenchimento do valor heurístico associado a cada subconjunto listado, tomando como base os valores heurísticos presentes nas dependências. Inicialmente, todas as posições do vetor de cálculo são inicializadas com valor igual a infinito, com excessão das posições do vetor que contêm elementos presentes na marcação sobre a qual o cálculo é realizado. Após o preenchimento, o valor heurístico é o máximo valor encontrado dentre as posições do vetor que referem-se aos lugares que pertencem ao conjunto de pré-condições da transição objetivo, referindo-se a uma estimativa da distância entre a marcação *m* e a marcação objetivo.

O pseudo-código apresentado no algoritmo 3.1 demonstra o cálculo da heurística para uma determinada marcação `m`. Apesar dos vetores de cálculo H^1 e H^2 possuirem complexidades diferentes, no que se refere ao número de elementos ou ao número médio de dependências, ambos compartilham a mesma estrutura básica, pois cada elemento do vetor de cálculo é associado a um conjunto de dependências que contém referências ao próprio vetor de cálculo. Por esta razão, o mesmo algoritmo é utilizado para o cálculo de ambas as heurísticas.

Algoritmo 3.1: *Método de cálculo da heurística*

```
1 input:Z[], m[], goal[];
2 output: integer;
3 function calculaHeuristica;
4         for(i = 0; i < size(Z); i = i + 1;)
5                 Z[i].heuristic = INFINITY;
```

```
6           endfor;
7           for(i = 0; i < size(m); i = i + 1;)
8                   Z[m[i]].heuristic = 0;
9           endfor;
10          flag = 0;
11          while(flag == 0)
12                  flag = 1;
13                  for(i = 0; i < size(Z); i = i + 1;)
14                  if(Z[i].heuristic = INFINITY)
15                          temp_min = INFINITY;
16                          for(j = 0; j < size(Z[i].c); j = j + 1;)
17                                  temp_max = 0;
18                                  for(k = 0; i < size(Z[i].c[j]); k = k + 1;)
19                                          temp_max = max(temp_max,
                                                           Z[i].c[j][k]);
20                                  endfor;
21                                  temp_min = min(temp_min, temp_max);
22                          endfor;
23                          if((temp_min + 1) < Z[i].heuristic)
24                                  Z[i].heuristic = temp_min + 1;
25                                  flag = 0;
26                          endif;
27                  endif;
28                  endfor;
29          endwhile;
30          heuristic = 0;
31          for(i = 0; i < size(goal); i = i + 1;)
32                  heuristic = max(heuristic, Z[goal[i]].heuristic);
33          endfor;
34          return heuristic;
```

As três funções pré-definidas do pseudo-código são:

- `size(v)`: Retorna o número de elementos encontrados no vetor `v`.
- `max(x,y)`: Retorna o maior valor dentre dois inteiros x e y.
- `min(x,y)`: Retorna o menor valor dentre dois inteiros x e y.

A linha (1) do algoritmo 3.1 apresenta os elementos de entrada para o cálculo da heurística sobre uma marcação `m`, cujos elementos são os seguintes:

- `Z`: Vetor de cálculo da heurística, cuja definição encontra-se na seção 3.1.2.
- `m`: Conjunto de condições que descreve o estado corrente da rede, sobre o qual a heurística é calculada.
- `goal`: Conjunto de condições que representa o estado objetivo da rede.

O tipo de saída do algoritmo é apresentado na linha (2), sendo um inteiro positivo que indica a distância entre os estados representados por `m` e `goal`.

O laço mostrado da linha (4) à linha (6) representa a inicialização do vetor de cálculo com *infinito*, neste caso, representado por um valor numérico alto. O objetivo de inicializar o vetor com o valor "infinito" é diferenciar os elementos em que o valor heurístico já foi calculado dos elementos onde este valor ainda não foi. Uma vez que o valor heurístico de um elemento qualquer tenha sido definido, este valor pode ser utilizado por outros elementos que o possuam em seus conjuntos de dependências.

Os algoritmos da família H^m calculam a heurística fazendo uma regressão do estado objetivo até o estado corrente. Da linha (7) à linha (9), todos os valores do vetor de cálculo cujos elementos pertençam ao estado corrente têm seu valor definido como zero, visto que, como já possuem uma marcação, não precisam de nenhum disparo de transição para serem habilitados. Neste ponto, todos os elementos do vetor de cálculo que referenciam condições marcadas da rede de Petri têm seu valor igual a zero e todos os elementos que referenciam condições não-marcadas tem valor igual a infinito. No caso do vetor de cálculo H^2, somente os elementos do vetor de cálculo $z_{p,q}$ em que ambas as condições `p` e `q` pertençam à marcação corrente é que terão o valor heurístico definido como zero.

O preenchimento do vetor de cálculo com os valores heurísticos adequados é apresentado das linhas (11) à (29). Na linha (13), um laço percorre todos os elementos do vetor de cálculo, e na linha (16), um laço percorre os elementos presentes nas suas respectivas dependências. Como já foi demonstrado na seção 3.1.2, cada elemento do vetor de cálculo possui várias dependências, sendo que cada uma delas é formada por uma regressão do subconjunto através de uma transição transcrita distinta. Assim, das linhas (19) a (21), é extraído o maior valor heurístico encontrado em cada uma das dependências presentes no vetor de cálculo, e das linhas (15) a (22), o menor valor encontrado entre o valor máximo de cada uma das dependências. O custo do subconjunto presente no elemento do vetor de cálculo é o valor mínimo encontrado dentre as dependências. O valor é acrescido de 1 na linha (24), pois uma regressão representa o estado do subconjunto antes do disparo de uma transição.

Uma única iteração pode não preencher todos os elementos do vetor de cálculo, pois se um elemento possuir uma dependência situada depois dele no vetor de cálculo, ele não será preenchido enquanto o valor da dependência não o for. Por este motivo, foi adicionada a variável `flag`, cuja função é verificar se durante a última iteração foi feita alguma atualização do vetor de cálculo. Caso uma iteração seja realizada e nenhuma modificação dos valores heurísticos seja feita, laço é encerrado.

O valor heurístico da marcação é extraído através do máximo valor encontrado nas posições do vetor de cálculo que referem-se ao estado objetivo, processo mostrado nas linhas (31) a (33).

Antes do preenchimento do vetor de cálculo, apenas os elementos do vetor de cálculo que referem-se à marcação corrente têm seus valores definidos, que no caso é o valor igual a zero. Depois da primeira iteração, garantidamente, todos os elementos do vetor de cálculo que não

tenham sido inicializados com zero, mas que possuam pelo menos uma dependência em que todos os elementos tenham valor igual a zero, terminarão com valor igual a 1. Assim, não existe a menor possibilidade de que, durante a execução da segunda iteração, qualquer elemento que não tenha seu valor definido na primeira iteração receba valor heurístico igual a 1. A cada nova iteração, os valores heurísticos atribuídos aos elementos do vetor de cálculo tendem a ser maiores do que os valores atribuídos nas iterações anteriores. Esta é a razão para a inserção do condicional presente na linha (14) do algoritmo. Uma vez que uma dependência tenha um valor máximo diferente de infinito, mesmo que todas as outras dependências ainda possuam o valor igual a infinito, na melhor das hipóteses as dependências retornarão um valor igual ao que já é conhecido.

Os valores preenchidos no vetor de cálculo são utilizados tanto para a identificação de subconjuntos inalcançáveis, descrito na seção 3.2.2, como para o próprio cálculo da heurística durante a busca, processo descrito na seção 3.2.3.

3.2.2 Otimizações provenientes da marcação inicial

A marcação inicial possui influência direta no problema de alcançabilidade de redes de Petri. Apesar do espaço de busca presente no problema de alcançabilidade ser igual a 2^n no pior caso, onde n representa o número de condições da rede de Petri, uma rede de Petri dificilmente alcança todas as marcações possíveis a partir da marcação inicial.

O preenchimento completo do vetor de cálculo tomando como base a marcação inicial da rede de Petri ($\mathtt{m}_0$) pode fornecer informações úteis sobre a alcançabilidade, visto que todos os subconjuntos são enumerados no vetor. Caso um subconjunto do vetor de cálculo seja alcançável a partir da marcação inicial, o valor heurístico atribuido a ele vai ser modificado.

Seja uma rede de Petri $\mathtt{N = <P, T, F>}$, associada a uma marcação inicial $\mathtt{m}_0$. Depois do preenchimento completo do vetor de cálculo H^1 a partir da marcação inicial $\mathtt{m}_0$, qualquer elemento $\mathtt{z}_p$ que permaneça com valor heurístico igual a infinito refere-se a um lugar da rede de Petri que nunca recebe uma marca, para qualquer marcação alcançável a partir de $\mathtt{m}_0$. No caso do vetor H^2, um elemento $\mathtt{z}_{p,q}$ com valor igual a infinito indica que não existe nenhuma marcação alcançável a partir de $\mathtt{m}_0$ em que as condições $\mathtt{p, q} \in \mathtt{P}$ sejam verdadeiras ao mesmo tempo. Como todos os subconjuntos cuja marcação seja igual a infinito são inalcançáveis a partir da marcação inicial, não é necessário tentar calcular a heurística de nenhum destes subconjuntos em qualquer marcação futura, da mesma forma como pode-se desconsiderar qualquer dependência que contenha um destes subconjuntos.

Outra otimização que é possível a partir do preenchimento do vetor de cálculo sobre a marcação inicial diz respeito à ordem em que os elementos estão dispostos no vetor de cálculo. Esta ordem influencia diretamente o desempenho do cálculo da heurística, pois da heurística

de um elemento depende que, pelo menos, uma de suas dependências possua todos os valores definidos. Entretanto, a ordem em que os subconjuntos estão dispostos no vetor de cálculo tem relação apenas com a ordem em que as condições são descritas no arquivo de entrada do *Mole*, ordem esta que dificilmente tem qualquer relação com a ordem de precedência das condições da rede.

A fim de otimizar o desempenho do cálculo da heurística, o vetor de cálculo é ordenado de acordo com os valores calculados a partir da marcação inicial da rede. Ainda que a ordem das heurísticas dos subconjuntos varie bastante ao longo da busca, a ordem obtida a partir do preenchimento sobre a marcação inicial respeita a precedência dos subconjuntos de uma maneira muito mais eficaz que a ordem aleatória encontrada no arquivo de entrada do *Mole*. Com isso, o número total de iterações necessárias para a obtenção do valor heurístico diminui, o que reduz também o tempo total de cálculo da heurística.

3.2.3 Processo de busca pelo plano ótimo

Nesta seção, será apresentada a utilização do algoritmo de busca A*, apresentado na seção 2.1.5, associado à ferramenta de desdobramento de redes de Petri *Mole*. Como foi tratado na seção 2.4, durante a execução do algoritmo de desdobramento de redes de Petri, os estados alcançáveis a partir da marcação inicial são armazenados em uma rede de ocorrências. As expansões da rede são feitas a partir de uma lista de eventos ainda não expandidos, ordenada de acordo com o tamanho da configuração local do evento. A expansão de um evento implica em retira-lo da lista de eventos, inseri-lo na rede de ocorrências, juntamente com as pós-condições da transição e verificar todas as transições habilitadas a partir desta nova marcação, que serão inseridas na lista de eventos não expandidos. Também é possível conhecer a marcação resultante do disparo da transição que é mapeada pelo evento, pois caso a marcação resultante já esteja presente na rede de ocorrências, então o evento é um evento de corte, que não será expandido.

Seja uma rede de Petri definida por `N = <P, T, F>`, considere a figura 3.2 que representa uma parte de um grafo de alcançabilidade de uma rede de Petri. Nesta figura, `s` representa o estado corrente da rede, ou seja, $s \subseteq P$. Os símbolos `<t', t", t"' ∈ T>` representam três transições habilitadas a partir do estado `s`. Os símbolos `s'`, `s"` e `s"'` representam três estados distintos resultantes dos disparos de suas respectivas transições, ou seja: `s[t'>s'`, `s[t">s"` e `s[t"'>s"'`.

Imagine que o estado `s` corresponda à marcação de um evento `e`, expandido pela ferramenta *Mole*. Ao ser expandido o evento `e`, as três transições `t'`, `t"` e `t"'` vão dar origem a três eventos distintos: E_n`(t')`, E_{n+1}`(t")` e E_{n+2}`("')`, cada qual associado com sua respectiva marcação. No momento em que estes eventos são gerados, é calculado o valor heurístico sobre a marcação associada a cada um deles (`s'`, `s"` e `s"'`), utilizando o algoritmo descrito na seção 3.2.1. Ao

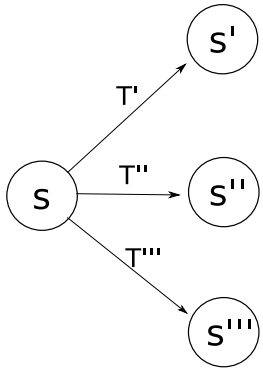


Figura 3.2: Fragmento de um grafo de alcançabilidade

valor obtido pelo cálculo da heurística, é somado o tamanho da configuração local dos eventos, visto que o tamanho da configuração local é igual ao custo de transformação da marcação inicial até a marcação associada ao evento. Se o evento tratar-se de um evento de corte, então ele não faz parte da solução do problema. Por este motivo, ao invés de ser calculada a heurística para a sua marcação, o valor heurístico recebe 1 000, a fim de que este evento seja inserido no fim da lista de prioridades de eventos. O valor 1 000 foi escolhido simplesmente por ser um valor alto. Nos testes realizados, nenhum problema teve soluções com um número tão grande de transições e a somatória do custo de obtenção do evento com o valor heurístico nunca vai ultrapassar o custo do plano ótimo, tendo como base heurísticas admissíveis e monotônicas.

A aplicação do algoritmo de busca A* no processo de desdobramento é feita através da mudança do parâmetro na ordenação da lista de eventos a serem expandidos. Ao invés de ordenar lista de eventos a serem expandidos com base apenas na configuração local, a lista é ordenada de acordo com a somatória do tamanho da configuração local e o valor heurístico obtido a partir da marcação inicial do evento. Como as heurísticas H^1 e H^2 são heurísticas admissíveis, o processo de busca sempre vai retornar o plano ótimo.

Sendo $C(e_n)$ a configuração local de um evento e_n, o valor heurístico atribuído a um evento e_n é definido pela função $f(e_n) = g(e_n) + h(e_n)$, onde $g(e_n) = \sum_{e \in C(e_n)} cost(\varphi(e))$ e $h(e_n) = H^m(Mark(e_n))$.

Em planejadores regressivos, como o planejador *HSPr*, que é apresentado por [11], as heurísticas da família H^m podem ser calculadas apenas uma vez, visto que a busca é realizada na mesma direção em que a heurística é calculada, ou seja, do estado objetivo até o estado inicial. Com isso, a regressão sempre vai partir de um estado intermediário até o estado inicial, de maneira que a referência da condição de parada (estado s_0) não vai mudar ao longo da busca e os valores obtidos da regressão de qualquer subconjunto podem ser reaproveitados caso o subconjunto apareça em qualquer outro momento durante a busca pela solução. Entretanto, a busca pelo plano através da ferramenta *Mole* funciona como um planejador progressivo, visto que a busca é realizada partindo do estado inicial até o estado objetivo. Ao ser realizado o cálculo da

heurística, a condição de parada da regressão não vai mais ser o estado inicial, como ocorre no planejador *HSPr*, mas os estados intermediários da busca, que mudam constantemente. Como a condição de parada vai mudar a cada passo da busca, a heurística precisa ser recalculada.

Com o intuito de otimização de tempo, a ferramenta *Mole* não mantém a lista de prioridades completamente ordenada. Ao invés disso, é utilizado um algoritmo do tipo "dividir e conquistar", que apenas fornece a garantia de que o evento com menor configuração local estará inserido na primeira posição da lista de prioridades. Além disso, o *Mole* utiliza outros critérios para a realização da ordenação quando se depara com dois eventos com o mesmo tamanho de configuração local, critérios estes baseados nas estruturas de dados utilizados pela ferramenta. Isto possibilita que um evento seja inserido na lista de prioridades à frente de outro evento criado anteriormente e que tenha o mesmo tamanho de configuração local. Esta característica do *Mole* destoa um pouco do algoritmo clássico de busca A*, visto que a ordenação da lista de prioridades de nós não-expandidos pelo algoritmo A* é feita de acordo com a função *f(x)* e de acordo com a ordem em que os nós foram criados. Na versão modificada do *Mole* para implementar o algoritmo A*, este algoritmo de ordenação foi mantido, visto que, como garante-se que os eventos com menor valor heurístico são expandidos primeiro, mantém-se a garantia de retorno do plano ótimo. Além disso, como a ordenação não se preocupa em ordenar toda a lista de prioridades, o processo é muito mais rápido do que se fosse feito de acordo com os critérios do algoritmo clássico de busca A*, tornando-se ineficiente quando o tamanho da lista de prioridades cresce muito.

Outra característica do *Mole* original é que o teste que verifica se a transição mapeada no evento é a transição objetivo é realizado no momento em que o evento é criado, e não quando é retirado da lista de prioridades, tal como ocorre no algoritmo clássico de busca A*. Uma vez que a transição corresponda ao objetivo, toda a lista de prioridades é eliminada, a fim de garantir que o evento que contém a transição objetivo seja inserido na primeira posição da lista. Consequentemente, o evento que contém a solução é expandido e o algoritmo de desdobramento é encerrado. Sem dúvida, a realização do teste de objetivo logo que o evento é criado diminui o tempo de busca, mas elimina a garantia de extração do plano ótimo, como pode ser verificado em um caso de teste apresentado na seção 4.2.2. Por esta razão, esta característica foi retirada, de maneira que o teste de objetivo da transição passou a ser realizado apenas quando o evento é retirado da primeira posição da lista de prioridades, que garantidamente possui a configuração local menor ou igual a qualquer outro evento presente na lista de prioridades, garantindo-se assim a solução ótima.

Entende-se por plano a sequência de transições ordenada que transforma a marcação inicial da rede de Petri na marcação objetivo, que é definida como o conjunto de pré-condições da transição objetivo. Ou seja, a transição objetivo é inserida apenas com o intuito de encerrar o desdobramento assim que ela for disparada, não fazendo parte diretamente do plano. Na reali-

dade, a real solução do problema é o evento que, depois de expandido, habilitou a geração do evento que contém a transição objetivo. Por este motivo, não faz sentido atribuir ao evento que contém a transição objetivo o valor heurístico calculado sobre a sua marcação correspondente, ou ainda o tamanho da configuração local, pois neste caso, seria necessário expandir todos os eventos no nível do evento anterior a este para que a busca seja encerrada.

Neste trabalho, o valor atribuído ao evento que contém a transição objetivo é o tamanho da configuração local do evento, subtraído de 2. Este valor foi definido por dois motivos: em primeiro lugar, o fato do evento que contém a transição objetivo não fazer parte do plano. Em segundo lugar, porque a solução já tinha sido encontrada em um nível anterior ao nível do evento com a transição objetivo. Como a heurística trabalhada é monotônica e admissível, se a primeira posição da lista de prioridades contiver um evento com o tamanho da configuração local igual ao tamanho da configuração local do evento subtraído de 2, na melhor das hipóteses vai encontrar uma solução com o mesmo número de transições que a solução já disponível. Com isso, o evento que mapeia a transição objetivo é inserido na posição mais à frente possível na lista de prioridades.

3.2.4 Extração do plano ótimo

Como foi apresentado na seção 3.2.3, a aplicação das heurísticas H^1 e H^2 no processo de desdobramento da rede de Petri não objetiva o prefixo completo da rede, mas apenas uma parte da rede de ocorrências que contenha a solução. O processo de desdobramento é encerrado assim que for feita a expansão do evento que se refere à transição objetivo, cujo pré-conjunto corresponda à marcação objetivo do problema.

A rede de ocorrências gerada até o momento em que a transição objetivo é disparada certamente contém o plano que é a solução do problema, mas também contém outros estados que não fazem parte da solução. Conforme apresentado na seção 2.4.3, a configuração local de um evento é a configuração mínima à qual um evento pode estar inserido. Por esta razão, é extraído da rede de ocorrências gerada pelo processo de desdobramento a configuração local do evento que encerrou o desdobramento. A extração do plano da configuração local é linear em relação ao tamanho do plano, e também possui a vantagem de retornar o plano ordenado.

3.3 Exemplo de cálculo

Esta seção mostra um exemplo de aplicação das heurísticas H^1 e H^2 em uma rede de Petri. O objetivo deste exemplo é uma demonstração minunciosa do uso das estruturas de dados utilizadas na construção do vetor de cálculo, bem como na dinâmica do uso das heurísticas para guiar o processo de busca A*. A apresentação de resultados experimentais obtidos a partir de

exemplos mais robustos é feita no capítulo 4.

A figura 3.3 mostra um exemplo de rede de Petri. O estado objetivo desta rede é representado pelo conjunto de condições {*p3*, *p5*}. A transição objetivo é a transição *GOAL*, cuja pré-condição corresponde à marcação objetivo da rede.

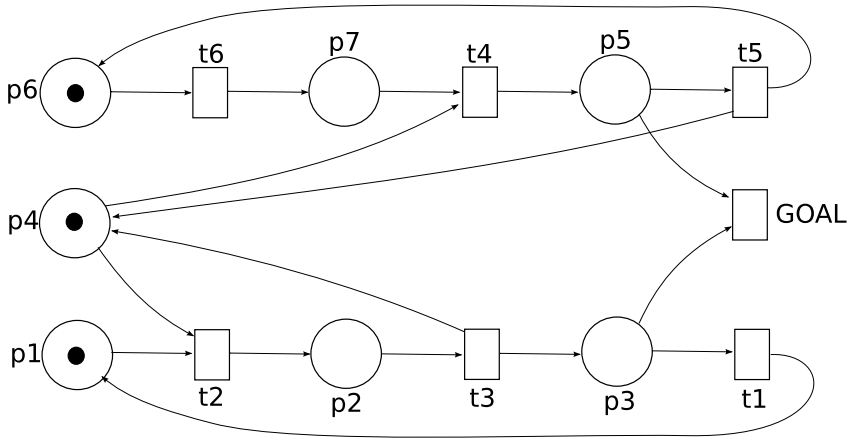


Figura 3.3: Exemplo de rede de Petri

Conforme foi apresentado na seção 3.1.1, a transcrição de transições em forma *STRIPS* é feita com o intuito de possibilitar a construção do vetor de cálculo, visto que as heurísticas H^1 e H^2 foram desenvolvidas para o ambiente de planejamento clássico. A tabela 3.1 mostra as transições transcritas da rede de Petri da figura 3.3.

Transição	**$^\bullet$t**	**t$^\bullet$**	**PRE**	**ADD**	**DEL**
GOAL	p3, p5	-	p3, p5	-	p3, p5
t1	p3	p1	p3	p1	p3
t2	p1, p4	p2	p1, p4	p2	p1, p4
t3	p2	p3, p4	p2	p3, p4	p2
t4	p4, p7	p5	p4, p7	p5	p4, p7
t5	p5	p4, p6	p5	p4, p6	p5
t6	p6	p7	p6	p7	p6

Tabela 3.1: Transições transcritas em forma *STRIPS*

Como a busca realizada pelo *Mole* através da heurística H^1 é diferente da busca realizada através da heurística H^2, as buscas serão apresentadas em duas seções distintas. A seção 3.3.1 apresenta a busca através da heurística H^1 e a seção 3.3.2 apresenta a busca através da heurística H^2. Em ambos os casos, será mostrado o preenchimento iterativo para a marcação inicial da rede de Petri da figura 3.3, a fim de identificar os subconjuntos inalcançáveis e realizar a ordenação de acordo com os valores atribuídos, processo descrito na seção 3.2.2. Em seguida, será demonstrado o desdobramento da rede de Petri com o auxílio do algoritmo de busca A*, conforme foi tratado na seção 3.2.3. Finalmente, será apresentada a extração do plano ótimo inserido na rede de ocorrências gerada, conforme apresentado na seção 3.2.4.

3.3.1 Heurística H^1

A tabela 3.2 mostra o vetor de cálculo H^1 da rede de Petri apresentada na figura 3.3, representando os lugares e as suas respectivas dependências. Como foi tratado na seção 3.1, essas dependências são geradas através da regressão dos subconjuntos a partir de uma transição transcrita em forma *STRIPS*. A condição *p1* contém uma única dependência, pois a única transição sobre a qual é possível fazer a regressão de *p1* é a transição *t1*. Esta dependência possui apenas um elemento, pois o conjunto *pre* da transição *p1* é composto por um único elemento, no caso, o elemento *p3*. Da mesma forma, a condição *p2* também possui apenas uma dependência, mas esta possui dois elementos, já que o conjunto *pre* da transição *t2* é composto pelos lugares *p1* e *p4*. No caso da condição *p4*, existem duas dependências, pois esta condição pode ser gerada pelas transições *t3* e *t5*, cujos conjuntos *pre* correspondem aos lugares *p2* e *p5*, respectivamente.

Conforme foi discutido na seção 3.2.1, o cálculo da heurística é realizado extraindo-se o menor valor dentre os valores heurísticos das dependências. O valor de cada dependência é o máximo valor encontrado em seus elementos. Desta forma, a heurística extraída do lugar *p2* é o maior valor encontrado dentre os lugares *p1* e *p4*, visto que ambos pertencem à mesma dependência. Por outro lado, a heurística extraída do lugar *p4* é o menor valor encontrado dentre os lugares *p2* e *p5*, pois estes pertencem a dependências distintas.

	Condição	**Depend.**	**it-0**	**it-1**	**it-2**	**it-3**
1	p1	3	0	0	0	0
2	p2	1, 4	∞	1	1	1
3	p3	2	∞	2	2	2
4	p4	2	0	0	0	0
		5				
5	p5	4, 7	∞	∞	2	2
6	p6	5	0	0	0	0
7	p7	6	∞	1	1	1

Tabela 3.2: Vetor H^1 do exemplo

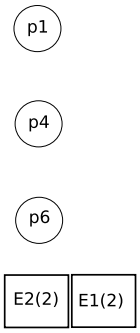
A tabela situada ao lado do vetor H^1 representa o preenchimento iterativo do vetor. A coluna *it-0* mostra o vetor H^1 antes do preenchimento iterativo. Todos os lugares marcados da rede são assinalados com 0 e todos os lugares não-marcados são assinalados com ∞. A coluna *it-1* mostra o vetor de cálculo após a primeira iteração do algoritmo mostrado na seção 3.2.1. A condição *p2* recebeu o valor heurístico igual a 1, pois tanto o lugar *p1* como o lugar *p4*, presentes nas suas dependências, foram inicializados com valor heurístico igual a zero. Uma vez que o valor de *p2* foi definido, é possível atualizar o valor de *p3*, pois este elemento possui uma dependência gerada pela transição *t3* cujo conjunto *pre* é formado pelo elemento *p2*. Entretanto, ao fim da primeira iteração, o valor da condição *p5* não pode ser atualizado, visto que a dependência deste elemento foi criada a partir da transição *t4*, cujo conjunto *pre* é formado pelas condições *p4* e *p7*, e a condição *p7* ainda não tinha sido atualizada quando foi

feita a tentativa de cálculo da heurística da condição *p5*. Na segunda iteração, a condição *p5* foi atualizada com valor igual a 2, visto que, na segunda iteração, todos os elementos de sua dependência estavam com valores heurísticos definidos. Como a terceira iteração não atualizou nenhum elemento, o laço é encerrado.

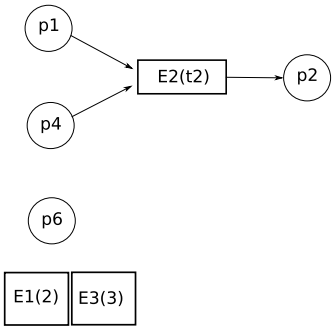
O estado objetivo da rede é representado pela pré-condição da transição *GOAL*, que é composta pelas condições *p3* e *p5*. Uma vez que o vetor H^1 tenha sido preenchido, o valor heurístico extraído a partir da marcação inicial da rede, no caso, o conjunto {*p1*, *p4*, *p6*}, é o máximo valor encontrado dentre os lugares que pertencem ao estado objetivo. Como o lugar *p3* possui valor heurístico igual a 2 e o lugar *p5* também possui valor heurístico igual a 2, o valor heurístico resultante do cálculo é 2.

Nesta rede de Petri, todos os elementos do vetor de cálculo terminaram marcados com valor heurístico diferente de ∞, o que significa que todos os lugares são alcançáveis a partir do estado inicial. Uma vez que não existem subconjuntos inalcançáveis, o vetor de cálculo pode ser ordenado tomando como base os valores heurísticos obtidos a partir da marcação inicial da rede, conforme apresentado na seção 3.2.2. Neste exemplo, a ordem dos elementos é a seguinte: {1, 4, 6, 2, 7, 3, 5}.

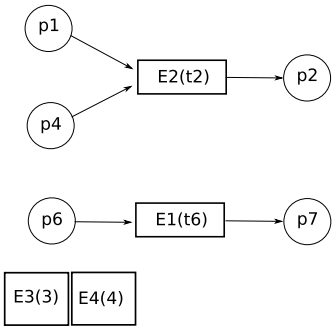
Neste ponto, as operações envolvendo a marcação inicial da rede de Petri já foram realizadas e é iniciado o algoritmo de desdobramento. Inicialmente, apenas as condições referentes à marcação inicial da rede de Petri são adicionadas à rede de ocorrência.



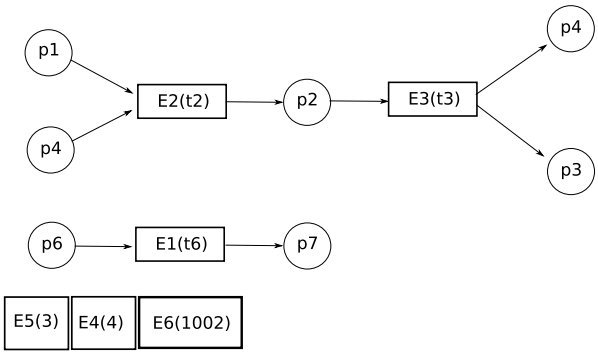
A lista de prioridades também é inicializada com as duas transições habilitadas a partir do estado inicial, que correspondem aos eventos *E1* e *E2*. O evento *E1* foi criado a partir da transição *t6* e habilita o estado {*p1*, *p4*, *p7*} e o evento *E2* foi criado a partir da transição *t2* e habilita o estado {*p2*, *p6*}. O cálculo heurístico a partir da marcação pertencente ao evento *E1* retorna valor igual a 2 e o cálculo a partir da marcação do evento *E1* também retorna valor igual a 2. Como a lista de prioridades está ordenada com o evento *E2* na primeira posição, o evento *E2* é expandido primeiro.



A expansão do evento *E2* gerou o evento *E3*, criado a partir da transição *t3*, que habilita a marcação {*p3*, *p4*, *p6*}. O valor heurístico obtido a partir desta marcação é igual a 2, mas como o evento *E3* já é precedido por um evento na rede de ocorrências, que no caso é o evento *E2*, é somado 1 à heurística. Desta forma, o próximo evento a ser expandido é o evento *E1*, que possui um valor heurístico menor.

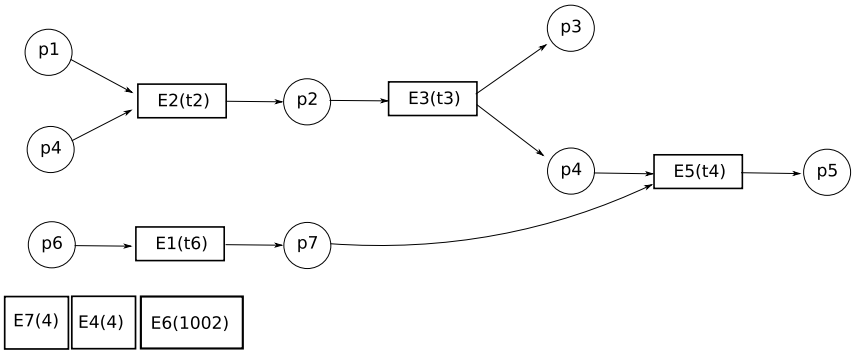


A expansão do evento *E1* gerou o evento *E4*, criado a partir da transição *t4*, que habilita o estado {*p1*, *p5*}. O valor heurístico obtido a partir desta marcação é igual a 3. A este valor, é somado 1, pois o evento é precedido pelo evento *E1*. O próximo evento a ser expandido é o evento *E3*.

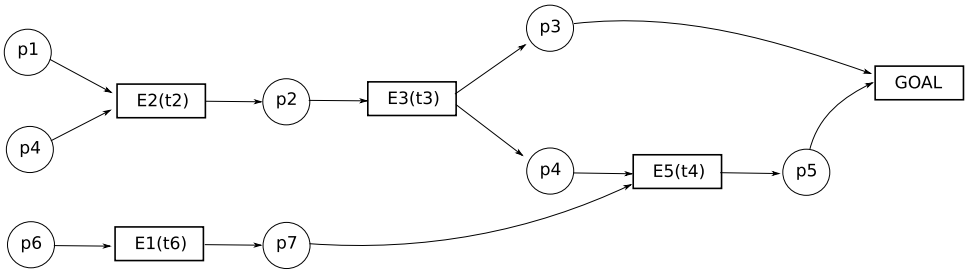


A expansão do evento *E3* gerou os eventos *E5* e *E6*. O evento *E5* foi criado a partir da transição *t4* e habilita a marcação {*p3*, *p5*} e o estado *E6* foi criado a partir da transição *t1* e habilita a marcação {*p1*, *p4*, *p6*}. O valor heurístico obtido a partir da marcação do evento *E5* retorna 0, pois todos os elementos pertencentes a esta marcação pertencem ao estado objetivo. A este valor, é somado 3, que é o tamanho da configuração local do evento *E5*. O evento *E6* é um evento de corte, pois a marcação de *E6* é idêntica à marcação inicial da rede. Por este

motivo, ao invés de calcular a heurística para este evento, simplesmente é atribuído o valor de heurística igual a 1 000, somado com o tamanho da configuração local do evento, que é igual a 2.



A expansão do evento *E5* gerou os eventos *E7* e *E8*. O evento *E7* foi criado a partir da transição *GOAL*, que corresponde à transição objetivo. Por este motivo, o valor heurístico do evento *E7* é definido como o tamanho da configuração local decrescido por um, visto que o plano não irá conter a transição *GOAL*. O evento *E8* é habilitado pela transição *t5* e habilita a marcação {*p3*, *p4*, *p6*}, que já foi encontrada por eventos anteriores, que o torna um evento *E8* um evento de corte.



A expansão do evento *E7*, que contém a transição *GOAL*, encerrou o desdobramento da rede de Petri. Os eventos não expandidos podem ser eliminados e pode-se extrair o plano ótimo da rede de ocorrências. A extração do plano ótimo consiste em enumerar todas as transições pertencentes à configuração local do evento que contém a transição objetivo. A transição objetivo não faz parte do plano, pois esta foi inserida na rede com o intuito de informar à ferramenta *Mole* que pode encerrar o desdobramento da rede de petri. A sequência de disparos de transições que transforma o estado inicial no estado objetivo corresponde ao seguinte conjunto de transições: {*t2*, *t3*, *t6*, *t4*}.

3.3.2 Heurística H^2

A tabela 3.3 mostra o vetor H^2 da rede de Petri mostrada na figura 3.3. Como foi apresentado na seção 3.1, as dependências são formadas a partir da regressão de um subconjunto de lugares a partir das transições transcritas, e contém referências ao próprio vetor de cálculo e não aos conjuntos resultantes da regressão. Como exemplo, considere o par de condições

p1-p2, que corresponde à linha 2 da tabela 3.3. Conforme tratado na seção 2.2.5, a regressão de um conjunto de condições envolve encontrar uma ação em que este conjunto possua interseção com o conjunto *add* e não possua interseção com o conjunto *del*. Ao se considerar as transições transcritas presentes na tabela 3.1, não existe nenhuma transição que gere simultaneamente os lugares *p1* e *p2*. A única transição que gera o lugar *p2* é a transição *t2*, mas como a condição *p1* está presente no conjunto *del* da transição *t2*, não é possível fazer uma regressão a partir de *t2*. A transição que gera a condição *p1* é a transição *t1*, que não contém a condição *p2* em seu conjunto *del*. Por esta razão, é possível realizar a regressão do par *p1-p2* a partir de *t1*, eliminando-se do subconjunto *p1-p2* todos elementos que pertencem ao conjunto *add* de *t1* (*p1*), e adicionando os elementos que pertencem ao conjunto *pre* de *t1* (*p3*), resultando assim no conjunto {*p2*, *p3*}. Este par de elementos possui uma referência à posição 9 do vetor de cálculo H^2, que passa a ser a dependência de *p1-p2*.

Outra regressão a ser destacada é a regressão das condições *p1-p5*, que corresponde à linha 5 do vetor de cálculo H^2. Não existe nenhuma transição que gere simultaneamente os lugares *p1* e *p5*. A transição que gera o lugar *p5* é a transição *t4*, que não possui o lugar *p1* contido em seu conjunto *del*. Por esta razão, a regressão do conjunto *p1-p5* a partir de *t4* pode ser realizada, resultando no conjunto {*p1*, *p4*, *p7*}. Este conjunto possui tamanho maior do 2, que é a ordem da heurística, o que obriga dividi-lo em subconjuntos de tamanho igual a 2. Os subconjuntos *p1-p4*, *p1-p7* e *p4-p7* apontam, respectivamente, para os elementos 4, 7 e 22 do vetor de cálculo. A dependência de *p1-p5* que referencia o índice 16 do vetor de cálculo refere-se ao conjunto *p3-p5*, gerado a partir da regressão pela transição *t1*.

No caso da regressão do conjunto *p3-p4*, existe uma transição que gera simultaneamente ambas as condições, que é a transição *t3*. Neste caso, a regressão do subconjunto *p3-p4* apenas enumera o conjunto *pre* da transição *t3*, que corresponde à condição *p2*. Como a condição *p2* possui apenas um elemento, a dependência passa a apontar para o índice 8, que corresponde ao conjunto *p2-p2*.

	Condições	**Depend.**	**it-0**	**it-1**	**it-2**	**it-3**
1	p1, p1	14	0	0	0	0
2	p1, p2	9	∞	∞	∞	∞
3	p1, p3	2	∞	∞	∞	∞
4	p1, p4	2	0	0	0	0
		5				
		15				
5	p1, p5	4, 7, 22	∞	∞	2	2
		16				
6	p1, p6	5	0	0	0	0
		17				
7	p1, p7	6	∞	1	1	1
		18				
8	p2, p2	4	∞	1	1	1

9	p2, p3	3, 4, 15	∞	∞	∞	∞
10	p2, p4	11	∞	∞	∞	∞
11	p2, p5	4, 5, 20	∞	∞	∞	∞
		10, 13, 22				
12	p2, p6	4, 6, 21	∞	1	1	1
		11				
13	p2, p7	4, 7, 22	∞	2	2	2
		12				
14	p3, p3	8	∞	2	2	2
15	p3, p4	8	∞	2	2	2
		16				
16	p3, p5	11	∞	∞	4	4
		15, 18, 22				
17	p3, p6	12	∞	2	2	2
		16				
18	p3, p7	13	∞	3	3	3
		17				
19	p4, p4	8	0	0	0	0
		23			0	0
20	p4, p5	11	∞	∞	∞	∞
21	p4, p6	12	0	0	0	0
		23				
22	p4, p7	13	∞	1	1	1
		25				
		21				
23	p5, p5	22	∞	2	2	2
24	p5, p6	21, 22, 27	∞	∞	∞	∞
25	p5, p7	24	∞	∞	∞	∞
26	p6, p6	23	0	0	0	0
27	p6, p7	25	∞	∞	∞	∞
28	p7, p7	26	∞	1	1	1

Tabela 3.3: Vetor H^2 do exemplo

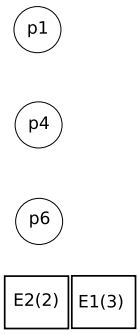
A tabela 3.3 demonstra o preenchimento iterativo do vetor H^2 para a marcação {*p1*, *p4*, *p6*}, que é a marcação inicial da rede de Petri. A coluna *it-0* representa o vetor de cálculo antes do cálculo da heurística. Em todos os pares onde ambos os lugares pertencem à marcação inicial, o valor é iniciado com zero. Nos demais pares, o valor é iniciado com infinito.

Da mesma maneira como ocorreu no vetor de cálculo H^1, o preenchimento iterativo a partir do estado inicial é realizado através de sucessivas iterações, até o momento em que uma iteração completa não modifica o valor heurístico de nenhuma posição do vetor de cálculo. No caso do vetor H^2, nem todas as posições do vetor de cálculo receberam valor heurístico diferente de ∞, como é o caso das condições *p2-p3*. Neste caso específico, observa-se na própria rede de Petri que a geração de uma marca em *p3* implica na eliminação de uma marca em *p2*. Por esta razão, as duas condições só poderiam estar marcadas simultaneamente se a condição *p3* possuir

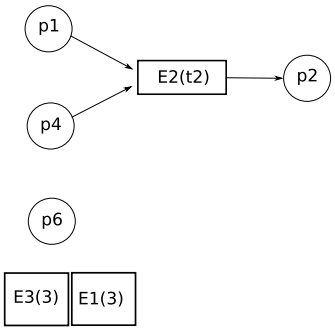
uma marca em um estado que habilite a transição *t2* para o disparo. O fato deste conjunto ter terminado com valor heurístico igual a ∞ implica que nenhuma marcação deste tipo é alcançável a partir da marcação inicial, ainda que, isoladamente, as condições *p2* e *p3* sejam alcançáveis, como foi demonstrado com o preenchimento do vetor de cálculo H^1, na tabela 3.2.

O estado objetivo da rede de Petri da figura 3.3 é composto pelo par de condições {*p3*, *p5*}, presente na posição 16 do vetor de cálculo H^2. Após a última iteração, o valor heurístico encontrado nesta posição é igual a 4, que é um valor maior do que 2, o encontrado pelo vetor de cálculo H^1.

Uma vez que tenham sido encontrados e eliminados do vetor de cálculo os subconjuntos inalcançáveis a partir da marcação inicial, este pode ser ordenado a partir dos valores encontrados pela heurística H^2. A nova ordem dos elementos do vetor de cálculo H^2 é a seguinte: {1, 4, 6, 19, 21, 26, 7, 8, 12, 22, 28, 13, 14, 15, 17, 23, 18, 16}. Nesta nova ordem, apenas 18 dos 28 elementos do vetor de cálculo estão listados, pois 10 dos 28 elementos constituem-se em subconjuntos inalcançáveis da rede de Petri.

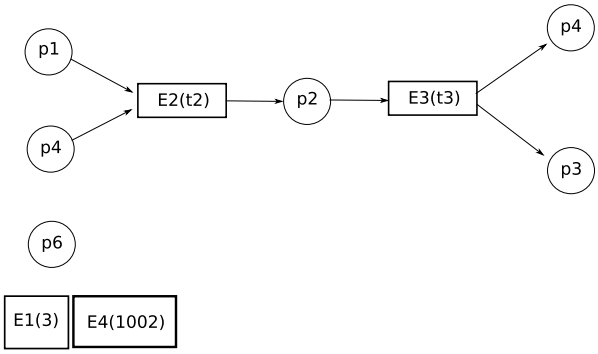


Da mesma maneira como aconteceu no desdobramento com o auxílio do vetor de cálculo H^1, dois eventos distintos são gerados a partir da marcação inicial: O evento *E2*, que é gerado pela transição *t2* e habilita a marcação {*p2*, *p3*} e o evento *E1*, que é gerado pela transição *t6* e habilita a marcação {*p1*, *p4*, *p7*}. A diferença é que o valor heurístico encontrado para a marcação do evento *E1* é igual a 3, em contrapartida ao valor encontrado pela heurística H^1, que é igual a 2.

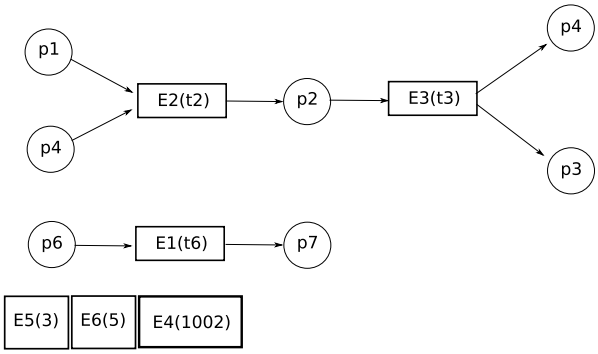


Após a expansão do evento *E2*, o evento *E3* é inserido na fila de eventos a serem expandidos. Este evento é gerado pela transição *t3* e habilita a marcação {*p3*, *p4*, *p6*}. O valor heurístico

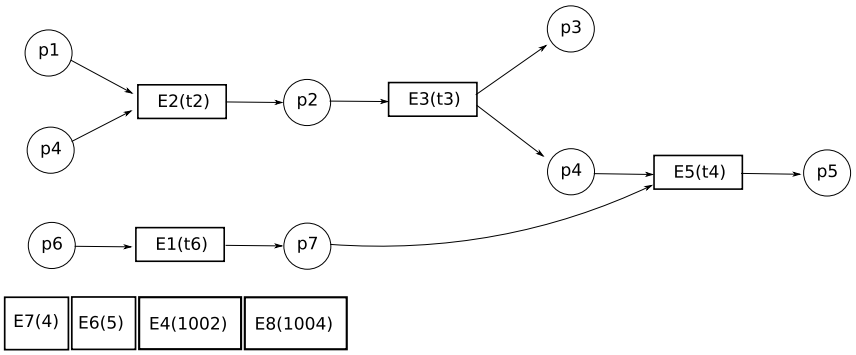
obtido a partir desta marcação é igual a 2, mas como o evento *E3* é precedido por um evento na rede de ocorrências, é somado 1 à heurística total. Por causa da ordenação de eventos da fila feita pelo *Mole*, tratada na seção 3.2.3, o evento *E3* é inserido à frente do evento *E1*.



A expansão do evento *E3* faz com que seja criado o evento *E4*, que é gerado pela transição *t1* e habilita o estado {*p1*, *p4*, *p6*}. Este estado corresponde ao estado inicial da rede de Petri, e por esta razão, este é um evento de corte. Por esta razão, o valor heurístico é definido como 1000, somado com número de eventos anteriores a ele. O próximo evento a ser expandido é o evento *E1*.

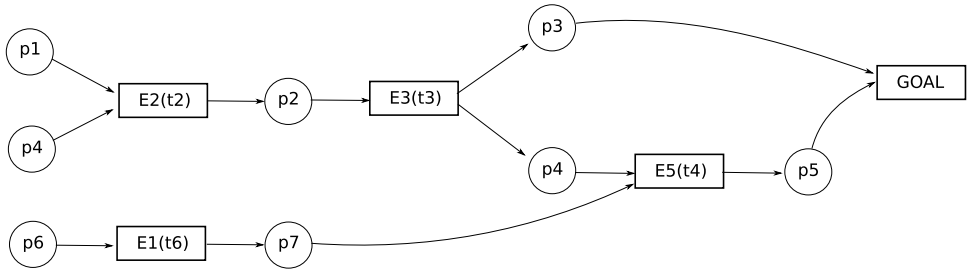


A expansão do evento *E1* faz com que os eventos *E5* e *E6* sejam criados. O evento *E5* é gerado pela transição *t4* e habilita a marcação {*p3*, p5} e o evento *E6* também é gerado pela transição *t4*, mas habilita a marcação {*p1*, *p5*}.



A expansão do evento *E5* faz com que os eventos *E7* e *E8* sejam criados. O evento *E8* é gerado pela transição *t5* e habilita a marcação {*p3*, *p4*, *p6*}. Como esta marcação já foi gerada anteriormente, o evento *E8* é um evento de corte. Por outro lado, o evento *E7* é gerado pela

transição *GOAL*, que é a transição objetivo do problema. Por isso, recebe como valor heurístico o tamanho de sua configuração local decrescido de 1, sendo inserido na primeira posição da fila de eventos a serem expandidos.



A expansão do evento *E7* encerra a execução do algoritmo de desdobramento. A extração do plano ótimo desta rede de ocorrências retorna o mesmo plano obtido através do desdobramento com auxílio da heurística H^1: {*t2, t3, t6, t4*}.

3.3.3 Relação entre as buscas entre H^1 e H^2

Conforme demonstrado nas seções 3.3.1 e 3.3.2, a utilização das heurísticas para guiar a busca não afeta diretamente o funcionamento do algoritmo de desdobramento de redes de Petri, tal como descrito em [7]. A única parte que foi modificada é a ordem em que os eventos estão dispostos na lista de prioridades de eventos não expandidos. Em virtude disso, algumas características do algoritmo de desdobramento que são úteis em um algoritmo de busca são mantidas.

A utilização do algoritmo de desdobramento possui um mecanismo próprio de *backtracking*, visto que estados repetidos são tratados como eventos de corte, sobre os quais não são realizadas expansões. Além disso, se o estado objetivo for inalcançável, o prefixo completo da rede também pode ser considerado como a condição de parada, de maneira que o algoritmo garantidamente vai parar a busca, sem o risco de entrar em um laço infinito. Na busca através do algoritmo A*, estas características podem não ser essenciais, visto que a busca A* é completa, mas são indispensáveis em um algoritmo que possui o risco de entrar em laço infinito, como é o caso do algoritmo de busca gulosa, utilizado pelo *GraphPlan*.

Na rede mostrada na figura 3.3, a utilização de heurísticas não foi relevante como acontece em redes maiores, tais como as redes utilizadas nos experimentos do capítulo 5. Isso aconteceu porque, nesta rede, são gerados poucos eventos a cada nova expansão da rede. No caso das redes tratadas no capítulo 5, são gerados tantos eventos a cada expansão que o tamanho da lista de prioridades cresce em uma proporção muito maior do que o crescimento da rede de ocorrências, conforme demonstraram os experimentos. Entretanto, esta demonstração já demonstra que a heurística H^2 retorna valores heurísticos melhores do que a heurística H^1. No preenchimento do vetor de cálculo a partir da marcação inicial, o vetor H^1 encontrou o valor heurístico igual a

2, enquanto que o vetor H^2 encontrou o valor heurístico igual a 4, quando o plano ótimo possui 4 disparos de transição. Em exemplos maiores, a consequencia direta é a redução do número de expansões da rede de Petri em cada busca.

A diferença da complexidade de cálculo entre as heurísticas H^1 e H^2 também mostrou-se evidente, especialmente no que tange ao tamanho dos vetores H^1 e H^2. A diferença na complexidade do cálculo só pode ser compensada se a diferença entre o número de expansões das heurísticas for muito menor no caso da busca pela heurística H^2.

3.4 Considerações

Neste capítulo, foram apresentadas as estruturas utilizadas que permitiram a adaptação das heurísticas de planejamento H^1 e H^2 ao contexto de desdobramento de redes de Petri. As estruturas de dados utilizadas permitiram que alguns detalhes das heurísticas da família H^m fossem construídos como pré-processamento, como é o caso das regressões dos subconjuntos de tamanho menor ou igual a m. Esta estrutura também permitiu que algumas otimizações fossem possíveis, como é o caso da identificação dos subconjuntos não alcançáveis. Como estes subconjuntos não são alcançáveis a partir da marcação inicial, eles podem ser ignorados durante o processo de busca, o que otimiza o cálculo da heurística posteriormente.

Ainda que apenas as heurísticas H^1 e H^2 tenham sido implementadas, a estrutura do vetor de cálculo permite que sejam implementadas futuramente heurísticas da família H^m de ordem mais elevada. Em 3.4, é apresentada a fórmula da heurística H^3, que também pertence às heurísticas da família H^m e que trabalha com conjuntos de tamanho menor ou igual a 3. Os itens (1), (2) e (3) são idênticos aos encontrados na heurística H^2. A diferença é o item (4), que mostra a regressão de um conjunto de tamanho 3, e o item (5), que divide o conjunto α em conjuntos β de tamanho igual a 3. Desta forma, a adaptação do vetor de cálculo para englobar a heurística H^3 demandaria apenas uma nova indexação do vetor de cálculo, a fim de que os conjuntos de tamanho igual a 3 também fossem listados e a implementação das regressões de conjuntos de tamanho igual a 3. As otimizações realizadas para a heurística H^2 e o próprio cálculo da heurística não seriam afetados.

$$
H^3(\alpha)=\begin{cases}
0, & \alpha \subseteq s_0 \quad (1)\\
\min\limits_{\alpha\subseteq add(a)}[c(a)+H^3(pre(a)], & |\alpha|=1,\alpha \not\subseteq s_0 \quad (2)\\
\min\limits_{\alpha=(p,q)}=\begin{cases}
\min\limits_{a\in L(p\&q)} & [c(a)+H^3(pre(a))] \quad (3.1)\\
\min\limits_{a\in L(p|q)} & [c(a)+H^3(pre(a)\bigcup\{q\})] \quad (3.2)\\
\min\limits_{a\in L(q|p)} & [c(a)+H^3(pre(a)\bigcup\{p\})] \quad (3.3)
\end{cases} & |\alpha|=2,\alpha \not\subseteq s_0 \quad (3)\\
\min\limits_{\alpha=(p,q,r)}=\begin{cases}
\min\limits_{a\in L(p\&q\&r)} & [c(a)+H^3(pre(a))] \quad (4.1)\\
\min\limits_{a\in L(p\&q|r)} & [c(a)+H^3(pre(a)\bigcup\{r\})] \quad (4.2)\\
\min\limits_{a\in L(p\&r|q)} & [c(a)+H^3(pre(a)\bigcup\{q\})] \quad (4.3)\\
\min\limits_{a\in L(q\&r|p)} & [c(a)+H^3(pre(a)\bigcup\{p\})] \quad (4.4)\\
\min\limits_{a\in L(p|q,r)} & [c(a)+H^3(pre(a)\bigcup\{q,r\})] \quad (4.5)\\
\min\limits_{a\in L(q|p,r)} & [c(a)+H^3(pre(a)\bigcup\{p,r\})] \quad (4.6)\\
\min\limits_{a\in L(r|p,q)} & [c(a)+H^3(pre(a)\bigcup\{p,q\})] \quad (4.7)
\end{cases} & |\alpha|=3,\alpha \not\subseteq s_0 \quad (4)\\
\max\limits_{\beta\subset\alpha,|\beta|=3}H^3(\beta), & |\alpha|>3,\alpha \not\subseteq s_0 \quad (5)
\end{cases}
\tag{3.4}
$$

4 *Resultados Experimentais*

Este capítulo trata dos resultados experimentais obtidos com a utilização das heurísticas H^1 e H^2 no processo de desdobramento de redes de Petri. As redes de Petri sobre as quais é realizada a busca são geradas pelo algoritmo Petrigraph [29], apresentado na seção 2.5.2, a partir de um conjunto de problemas de planejamento clássico. A seção 4.1 apresenta os problemas utilizados e a metodologia empregada na execução dos experimentos. A seção 4.2 apresenta a análise dos resultados obtidos para cada conjunto de problemas.

4.1 Metodologia

Os experimentos foram realizados a partir de redes de Petri geradas pelo Petrigraph, apresentado na seção 2.5.2. O Petrigraph gera redes de Petri cíclicas e seguras, que são equivalentes ao problema de planejamento sobre o qual foram criadas. Em todas as instâncias dos problemas, foi inserida a transição *GOAL*, cujas pré-condições equivalem à marcação objetivo.

As redes de Petri geradas a partir dos problemas de planejamento foram submetidas ao algoritmo de desdobramento orientado por uma versão modificada da ferramenta *Mole*, que integra as heurísticas H^1 e H^2 à busca, conforme foi descrito no capítulo 3.

Por causa de restrições de tempo e de memória computacional, todos os testes foram submetidos a um limite de tempo de 2 500 segundos. Os problemas nos quais o *Mole* encontrou uma solução no tempo limite foram analisados de acordo com os seguintes critérios:

- Tempo de execução.
- Número de expansões da rede de ocorrências.
- Número total de eventos gerados.

O tempo de execução dos problemas foi comparado com o tempo de execução do *SatPlan* [15] e com o tempo de execução utilizando a heurística implementada por Töws [29], que é descrita na seção 2.5.2. Ainda que a modificação da ferramenta *Mole* feita por Töws não

tenha como objetivo o plano ótimo, os números apresentados têm a função de comparar o tempo obtido pela implementação deste trabalho com outra modificação da ferramenta *Mole*. O *SatPlan* é um planejador que converte os problemas de planejamento em um problema de satisfabilidade de booleanos. Esta ferramenta possui a vantagem de retornar o plano ótimo, sendo portanto útil para efeitos de comparação de tempo com este trabalho, que também objetiva o plano ótimo.

A análise do número de expansões da rede de ocorrências gerada até o momento em que a busca é encerrada possui a função de demonstrar o tamanho do espaço de estados explorado para se obter o plano. A informação do número total de eventos gerados, por outro lado, possui a função de demonstrar o trabalho real realizado pelo *Mole* para obter a solução.

Também forma feitas análises sobre a complexidade do vetor de cálculo da heurística e a profundidade alcançada pelos problemas em que a solução não foi encontrada no tempo limite de 2 500 segundos. Em alguns domínios, foi feita uma análise mais detalhada de instâncias específicas de problemas, com o intuito de verificar como o *Mole* se comporta ao longo das expansões realizadas.

Os experimentos foram realizados em um computador com processador 2 Opteron Dual Core, com 32 Gb de memória RAM, com sistema operacional Linux. A mesma máquina foi utilizada para obter o tempo de execução dos problemas através dos planejadores *SatPlan* e *GraphPlan*.

4.2 Análise dos resultados

Esta seção trata da análise dos resultados obtidos a partir da solução dos problemas de alcançabilidade a partir das redes de Petri produzidas pelo Petrigraph. Os domínios utilizados nos experimentos foram propostos nas competições de planejamento AIPS-2000[1] e IPC-2002[16]. Este é o mesmo conjunto de testes submetido por Töws em [29], com o intuito de testar a eficiência do Petrigraph. A tabela 4.1 mostra o número de problemas solucionados em cada domínio. Nas seções seguintes, serão mostrados os dados referentes a cada problema trabalhado.

Durante a demonstração dos resultados experimentais, sempre que forem apresentados números relativos à heurística H^0, estes referem-se à execução do *Mole* utilizando a heurística mostrada na fórmula 4.1. Na heurística H^0, a ferramenta *Mole* comporta-se conforme o algoritmo de busca de custo uniforme, mantendo a característica de obtenção do plano ótimo.

$$H^0(\alpha) = 0 \tag{4.1}$$

Domínio	Total	H^0	H^1	H^2
Blocksworld	35	12	15	17
Logistics	28	10	10	10
Elevator	101	35	35	35
Driverlog	18	5	7	7
Rovers	20	4	5	5
Satellite	20	1	3	3
Zenotravell	20	6	8	8

Tabela 4.1: Número de problemas por domínio

O fato de ter sido utilizado um conjunto de testes baseado no ambiente de planejamento não significa que apenas problemas de planejamento transcritos em formato de rede de Petri possam ser solucionados com esta técnica. Como o objetivo deste trabalho é a solução de problemas de alcançabilidade de redes de Petri e não a solução de problemas de planejamento, todas as análises feitas terão como foco os aspectos próprios das redes de Petri e dos algoritmos de busca realizados para a obtençaõ da solução.

4.2.1 Blocksworld

O domínio *Blocksworld* descreve uma família de problemas baseada em operações com um conjunto de blocos sobre uma mesa. Os blocos podem ser empilhados, mas apenas um bloco pode ficar diretamente sobre outro. Os blocos podem ser movidos com o auxílio de uma garra, que movimenta um bloco por vez, podendo colocá-lo sobre outro bloco ou diretamente sobre a mesa. O objetivo do problema é, dada a configuração inicial dos blocos, movê-los para uma outra configuração.

A figura 4.1 mostra o número de lugares e transições das redes de Petri geradas pelo Petrigraph a partir dos problemas do domínio *Blocksworkd* e a figura 4.2 apresenta o tempo necessário para a obtenção da solução nos referidos problemas.

As redes de Petri de número 1 a 9, que possuem no máximo 60 lugares e 85 transições, tem o plano ótimo encontrado em um tempo inferior a 1 segundo, para qualquer heurística utilizada. A partir da rede de número 13, o tempo de execução possui um aumento significativo. Nas redes de número 13, 14 e 15, com 93 lugares e 145 transições, a heurística H^0 não encontra a solução ótima em um tempo inferior a 2 500 segundos. Para os problemas de número 16 e 18, com 117 lugares e 181 transições, apenas a heurística H^2 encontra a solução no tempo limite de 2 500 segundos. A partir do problema 19, com 140 lugares e 221 transições, nenhuma das heurísticas é eficiente o suficiente para encontrar uma solução em menos de 2500 segundos. Para os problemas pequenos, ou seja, com tempo inferior a 1 segundo, o uso da heurística H^2 não mostrou-se vantajoso em relação à heurística H^1 e até em relação à heurística H^0 em alguns problemas. Entretanto, à medida em que a complexidade dos problemas aumenta, a heurística

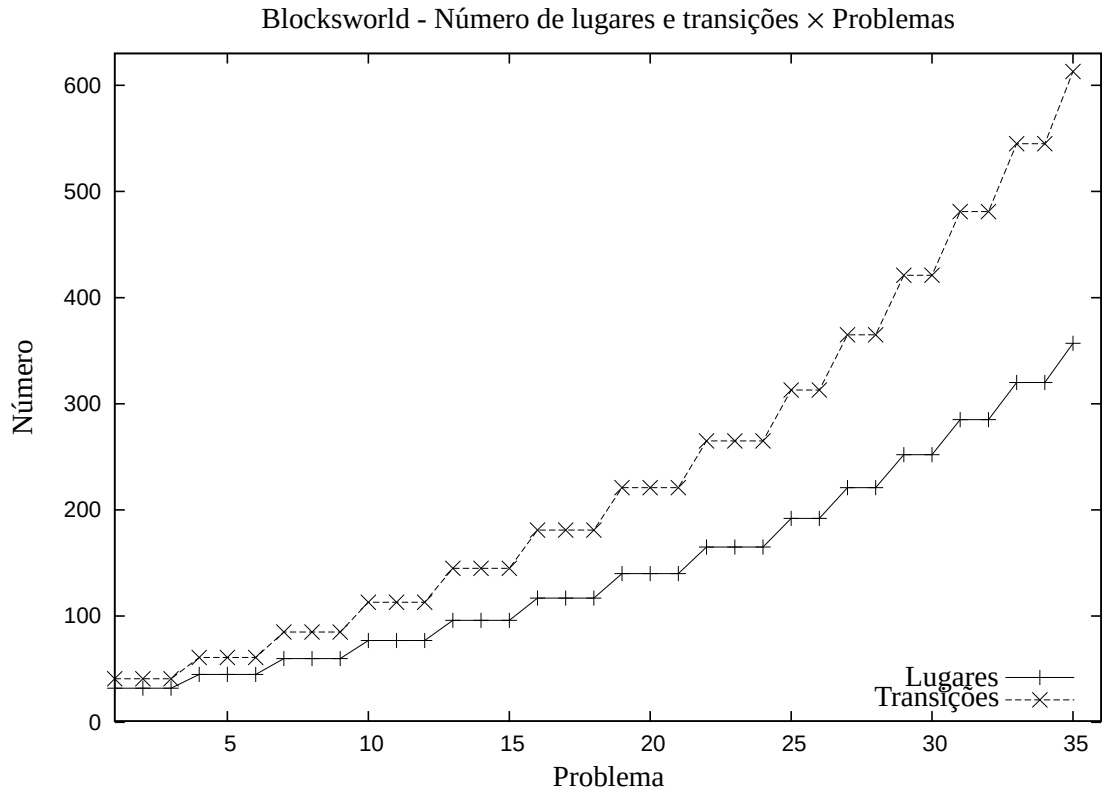


Figura 4.1: Número de Lugares e Transições das redes do domínio *Blocksworld*

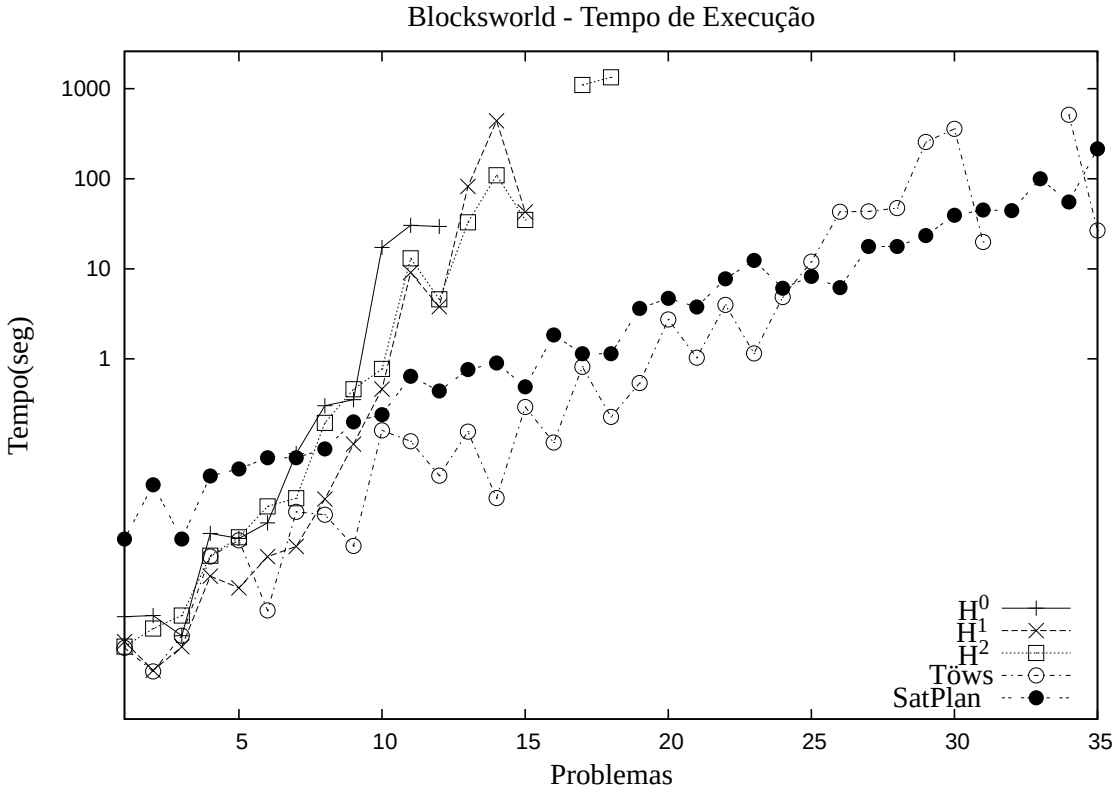


Figura 4.2: Tempo de execução do domínio Blocksworld

H^2 passou a ser mais vantajosa do que as heurísticas H^1 e H^0. Uma das razões que justificam isso é que a taxa de expansões das redes de Petri realizadas pelo *Mole* não é constante, conforme será demonstrado adiante.

O tempo de execução da busca orientada pela heurística de Töws foi menor do que o tempo de execução das heurísticas H^1 e H^2, na maioria dos casos, visto que a busca realizada por *Töws* não possui como objetivo o plano ótimo. A busca realizada pelo *SatPlan*, entretanto, mostrou-se menos eficiente nos problemas de pequeno porte, tornando-se mais atrativa à medida em que a complexidade dos problemas cresce. Percebe-se também que os problemas onde a busca realizada pelas heurísticas H^1 e H^2 foi melhor do que a busca realizada pelo *SatPlan* são justamente os problemas de pequeno porte onde a heurística H^2 possui um desempenho de tempo inferior à busca realizada pelas heurísticas H^0 e H^1.

O número de expansões necessárias para encontrar o plano ótimo através do algoritmo A* é mostrado na figura 4.3. Ao contrário do que ocorre em relação ao tempo de execução, o número de expansões realizadas pelo desdobramento da rede de Petri possui relação direta com a ordem das heurísticas. Em todos os casos analisados do domínio *Blocksworld*, a heurística H^2 encontrou a solução ótima, mas o número de expansões realizadas cresce exponencialmente à medida em que aumenta a complexidade do problema.

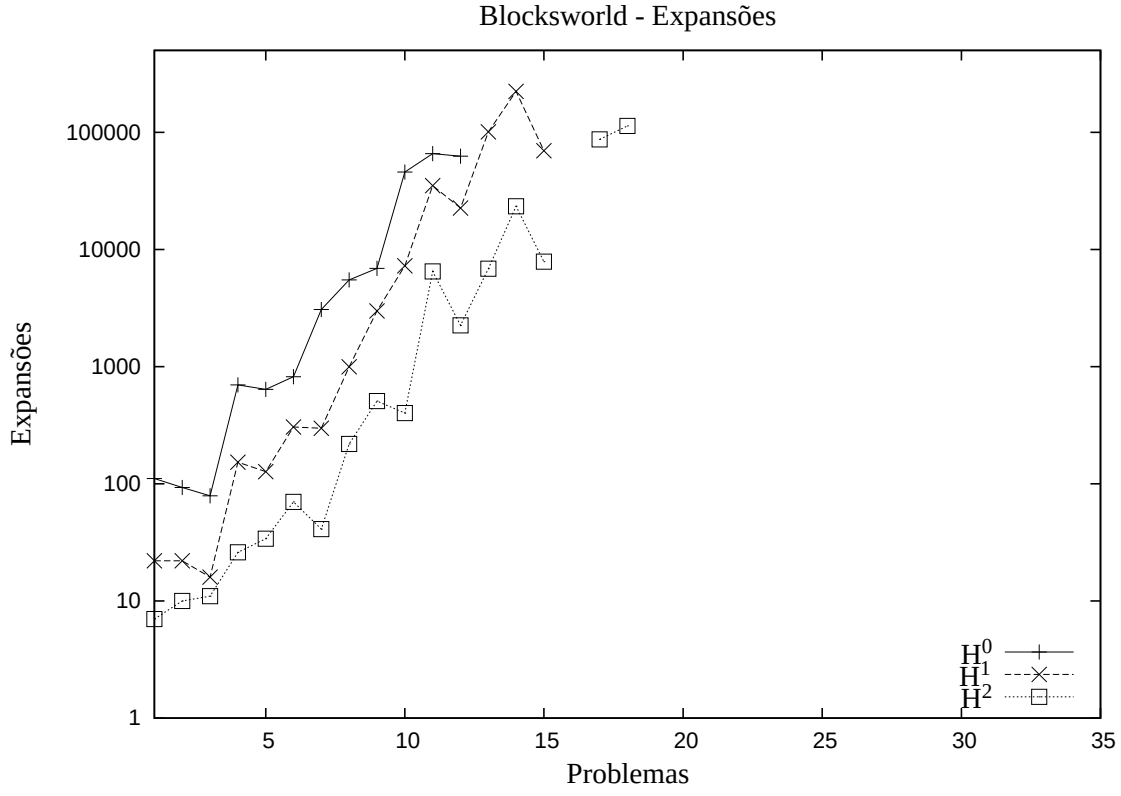


Figura 4.3: Número de expansões do domínio *Blocksworld*

O tempo unitário de uma expansão é influenciado pelo tamanho da rede de ocorrências

Expansões	Tempo (seg)
1 000	0.021403
2 000	0.075863
4 000	0.141071
8 000	0.688486
16 000	1.982957
32 000	10.025101
64 000	49.817356
128 000	234.183213
256 000	621.556693
512 000	3 508.642813
665 460	4 823.548855

Tabela 4.2: Relação expansões-tempo em uma busca não-orientada

gerada até o momento, pois para realizar uma expansão, o *Mole* procura na rede de ocorrências por lugares habilitados a fazer parte de uma expansão, no caso, lugares que não fazem parte do pré-conjunto de nenhuma transição. À medida que a rede de ocorrências cresce, o tempo necessário para procurar por lugares habilitados torna-se maior. Deste modo, quanto maior o tamanho da rede de ocorrências gerada até o momento, menor a taxa de expansões por segundo realizadas pelo algoritmo de desdobramento. Para demonstrar esta informação, observe a tabela 4.2. Nesta tabela, todos os números referem-se à execução do *Mole* para a mesma instância do problema com a heurística H^0, relacionando números e tempos de expansões parciais até ser encontrado o plano ótimo.

De acordo com os números da tabela 4.2, a realização de 64 000 expansões ocorreu em aproximadamente 50 segundos. Se a taxa de expansões fosse constante, a realização de 128 000 expansões deveria ser feita em aproximadamente 100 segundos. Entretanto, o tempo necessário para realizar 128 000 expansões é de 234 segundos. Ou seja, a taxa de expansões caiu mais do que pela metade se comparar as primeiras 64 000 expansões com as 64 000 expansões seguintes. O tempo necessário para encontrar o plano ótimo para este problema é de 4 823 segundos, executando para isso 665 460 expansões. Neste exemplo, a heurística H^1 encontrou a solução com 223 704 expansões, utilizando 456 segundos e a heurística H^2 encontrou a solução com 23 383 expansões, utilizando 134 segundos.

Essa influência que o tamanho da rede de ocorrências possui sobre a taxa de expansões do *Mole* é o principal fator que torna a busca guiada por heurísticas muito mais vantajosa do que a busca realizada pela versão original do *Mole*. Neste exemplo, a heurística H^0 realizou apenas três vezes mais expansões do que a heurística H^1, mas consumiu 10 vezes mais tempo. Já a heurística H^2 encontrou a solução com um número de expansões 20 vezes menor do que a heurística H^1, consumindo 3 vezes menos tempo. Mesmo que o cálculo da heurística H^2 seja muito mais lento do que o cálculo da heurística H^1, o número menor de expansões pesou bastante para que o tempo de execução da heurística H^2 seja menor do que o tempo da heurística

H^1.

Enquanto que o número de expansões refere-se apenas ao tamanho da rede de ocorrências gerada até o momento, os números apresentados na figura 4.4 referem-se ao número total de eventos criados pelo *Mole*, que é a soma do número de expansões, o número de eventos de corte e o tamanho da lista de prioridades de eventos não-expandidos.

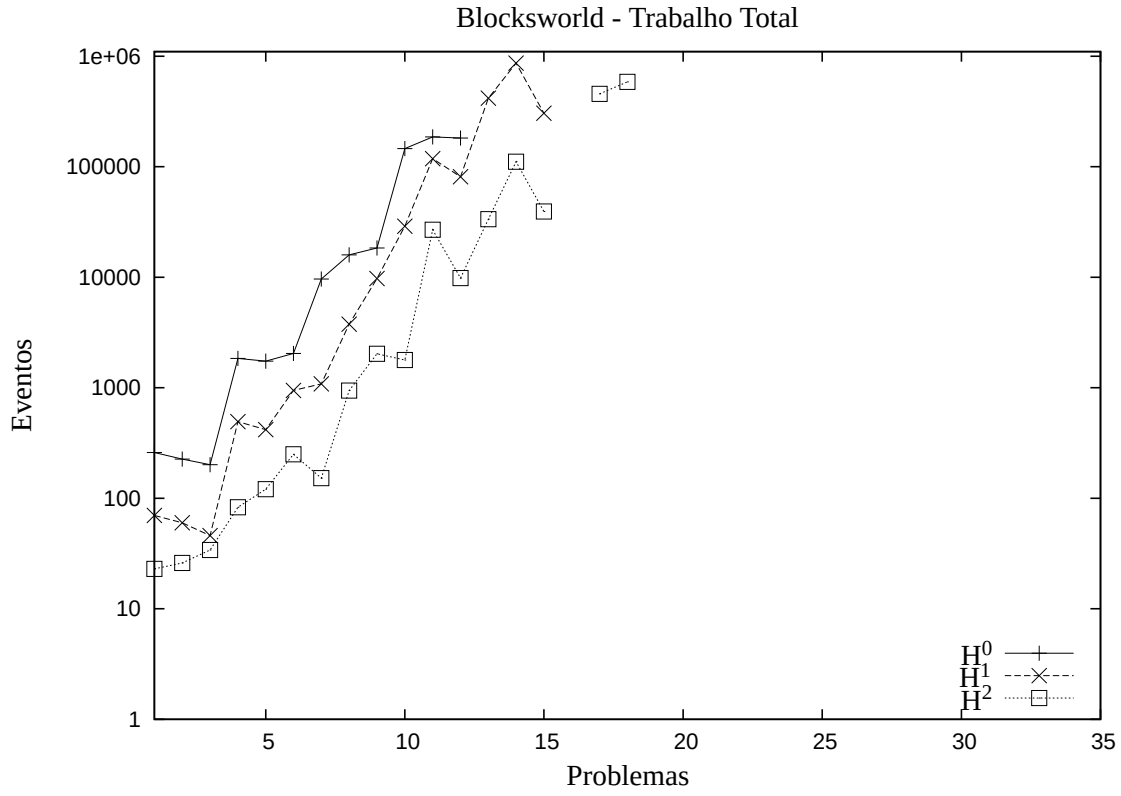


Figura 4.4: Eventos gerados no domínio *Blocksworld*

Outro dado que reflete o aumento da complexidade dos problemas é mostrado na figura 4.5, que refere-se ao número médio de dependências do vetor de cálculo. Quanto mais dependências cada elemento do vetor de cálculo possuir, mais complexo é o cálculo da heurística sobre a marcação referente ao evento, e consequentemente, menor o tempo necessário para a realização da expansão.

O aumento da complexidade do problema também reflete no aumento do tamanho do plano ótimo. Como as heurísticas foram utilizadas no contexto do algoritmo A*, todos os problemas que foram solucionados no tempo limite de 2 500 segundos resultaram no plano ótimo. A figura 4.6 mostra, para os problemas que não foram solucionados, a distância entre o tamanho do plano ótimo e o nível encontrado em cada um deles. Nestes problemas, o plano ótimo foi obtido a partir da ferramenta SatPlan.

Neste domínio, em todos os problemas onde a solução ótima não foi encontrada no tempo limite de 2 500 segundos, a heurística H^2 chegou mais perto da solução do que a heurística H^1,

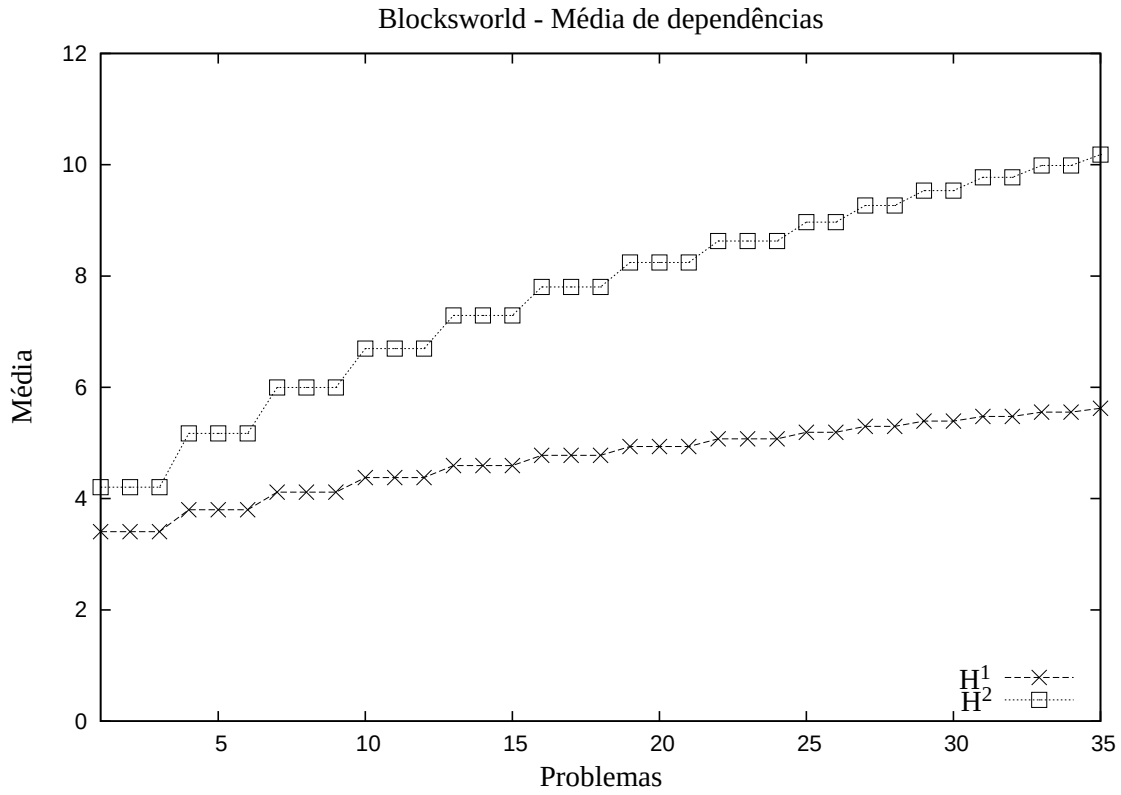


Figura 4.5: Número médio de dependências do domínio *Blocksworld*

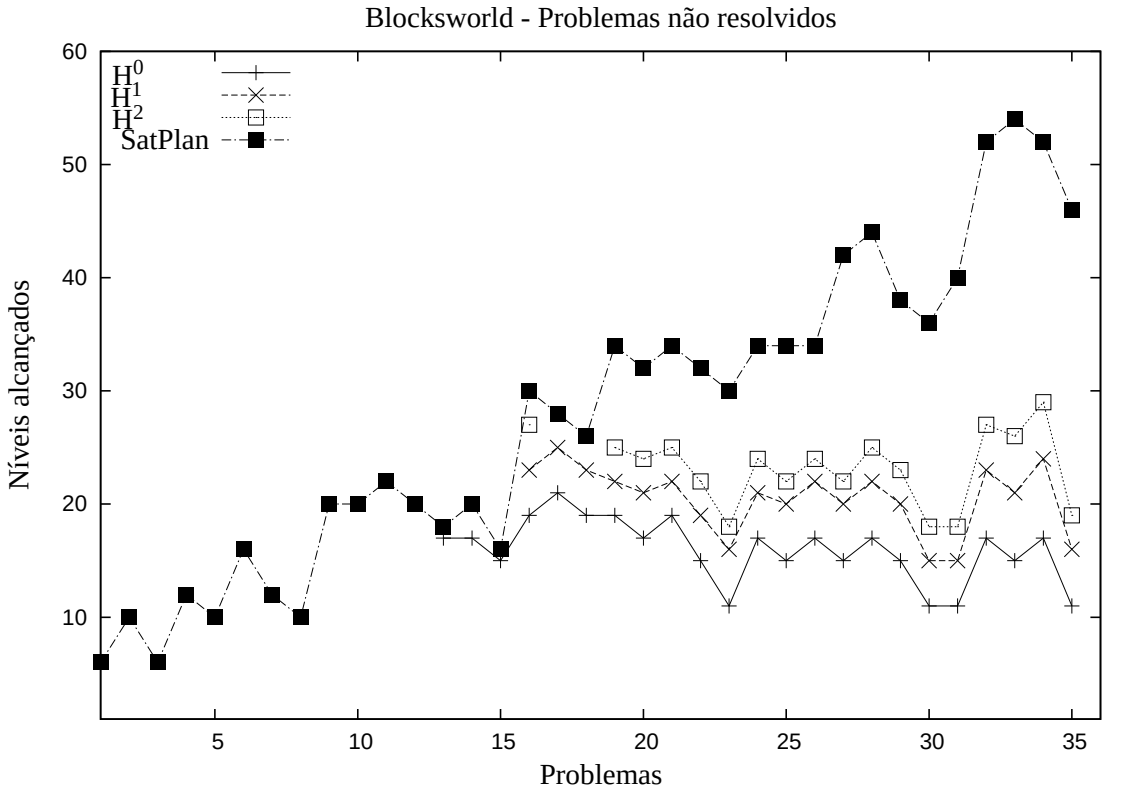


Figura 4.6: Profundidade explorada pelos problemas não-resolvidos no domínio *Blocksworld*

mesmo com o aumento da complexidade do vetor de cálculo H^2.

4.2.2 Driverlog

O domínio *Driverlog* define um problema de logística de pacotes. Neste domínio, os pacotes precisam ser transportados por caminhões entre diversas localidades. Os caminhões são dirigidos por motoristas, que podem estar em diversos lugares e precisam caminhar até o local onde está o caminhão para que o transporte possa ser realizado.

A figura 4.7 apresenta o número de lugares e transições das redes de Petri geradas a partir do domínio *Driverlog* e a figura 4.8 apresenta os tempos de execução para encontrar o plano ótimo.

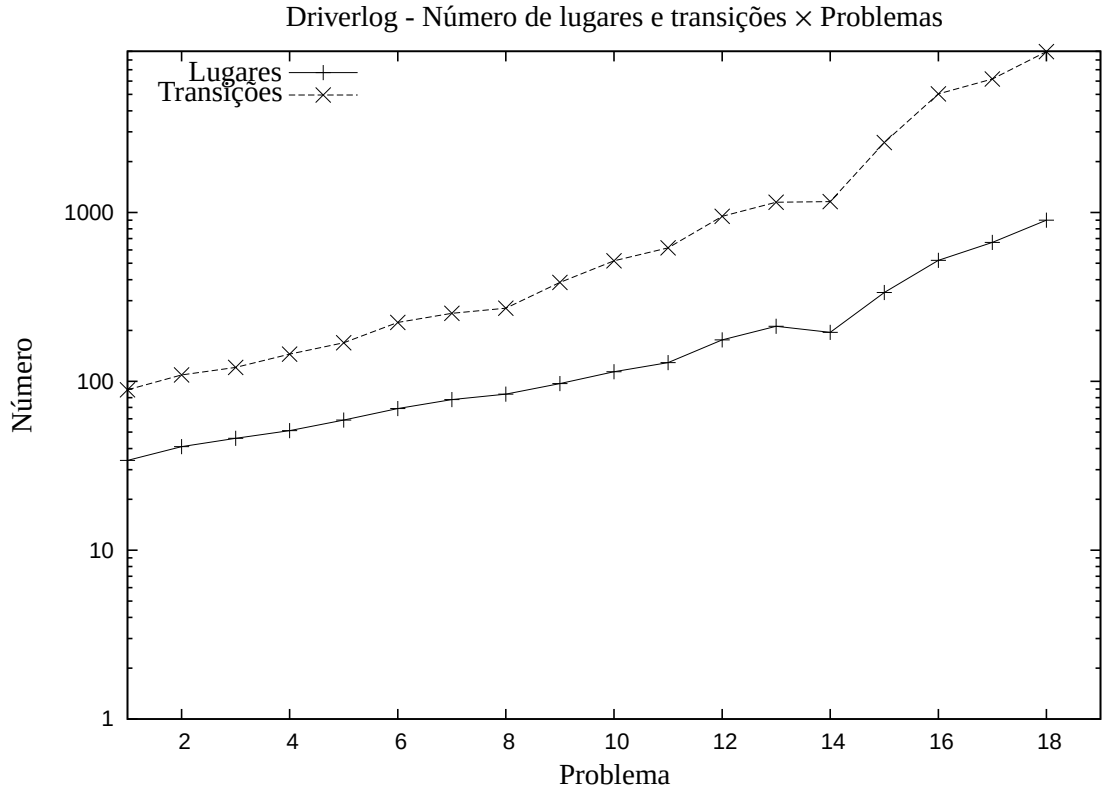


Figura 4.7: Número de Lugares e Transições das redes do domínio *Driverlog*

A complexidade das redes produzidas neste domínio é maior do que nas redes encontradas no domínio *Blocksworld*, no que se refere ao número de lugares e transições. Enquanto que no domínio *Blocksworld* o número de lugares varia entre 32 e 357 e o número de transições varia entre 42 e 613, o número de lugares das redes do domínio *Driverlog* varia entre 34 e 900 e o número de transições varia entre 89 e 8961. O tempo médio de execução para encontrar o plano ótimo também é maior, conforme apresenta a figura 4.8. Apenas no primeiro exemplo a heurística H^0 encontrou uma solução em tempo inferior a 1 segundo. Da mesma forma como

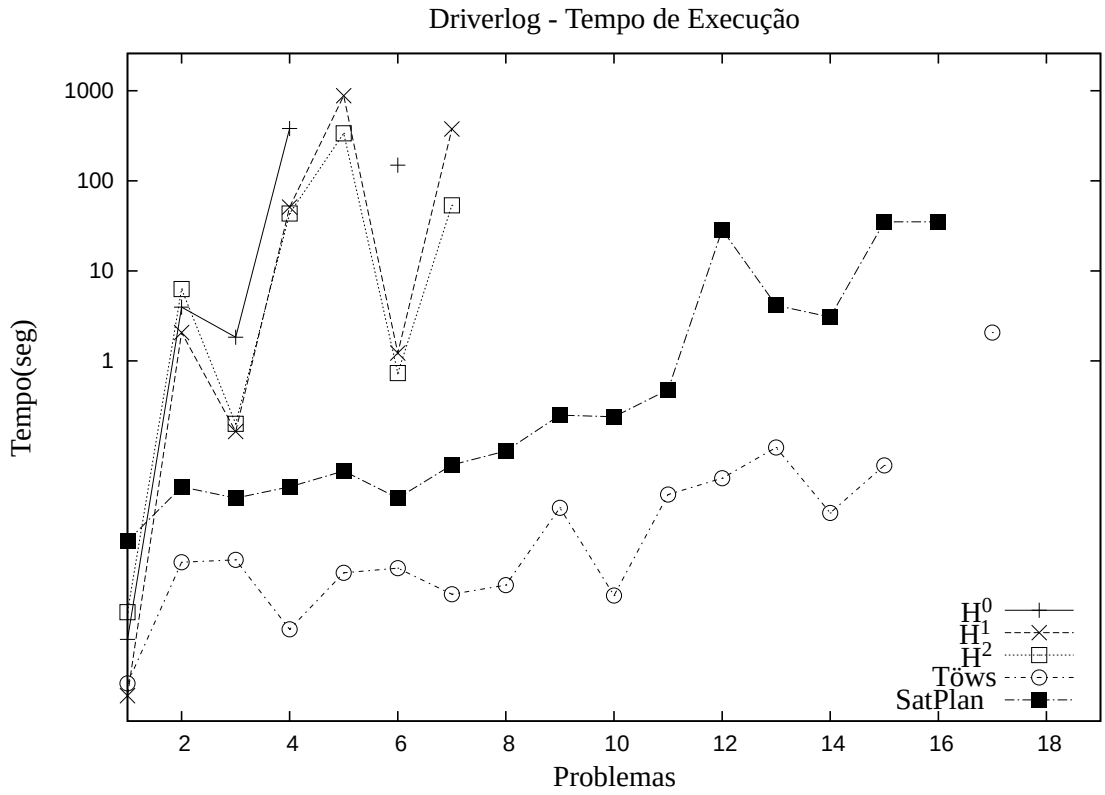


Figura 4.8: Tempo de execução do domínio Driverlog

no domínio *Blocksworld*, a heurística H^2 se mostrou menos eficiente do que a heurística H^1 em problemas de pequeno porte, se tornando mais eficiente à medida em que a complexidade dos problemas aumenta. A partir do problema 8, nenhuma das heurísticas encontra uma solução no tempo limite de 2 500 segundos.

O tempo de execução do *Mole* orientado pela heurística de Töws se mostou mais eficiente do que o tempo utilizado pelo *SatPlan*, e apenas no primeiro exemplo a busca orientada pela heurística H^1 encontrou a solução em tempo ligeiramente menor do que a heurística de Töws. Vale ressaltar que na busca orientada pela heurística de Töws, o teste de solução é realizado quando o evento é criado, e a busca é encerrada com a primeira solução encontrada. A busca implementada neste trabalho, por sua vez, procura obter o plano ótimo, que nem sempre corresponde à primeira solução encontrada.

O aumento da proporção entre transições e lugares refletiu no número de expansões necessárias para encontrar o plano ótimo, apresentado na figura 4.9. Assim como ocorreu com o domínio *Blocksworld*, a heurística H^2 se mostrou mais eficiente do que a heurística H^1 no que diz respeito ao número de expansões.

O trabalho total realizado é apresentado na figura 4.10. A proporção de eventos gerados para cada expansão do domínio *Driverlog* é muito maior do que a encontrada no domínio *Blocksworld*. O problema 5, por exemplo, gerou mais de 4 milhões de eventos para realizar

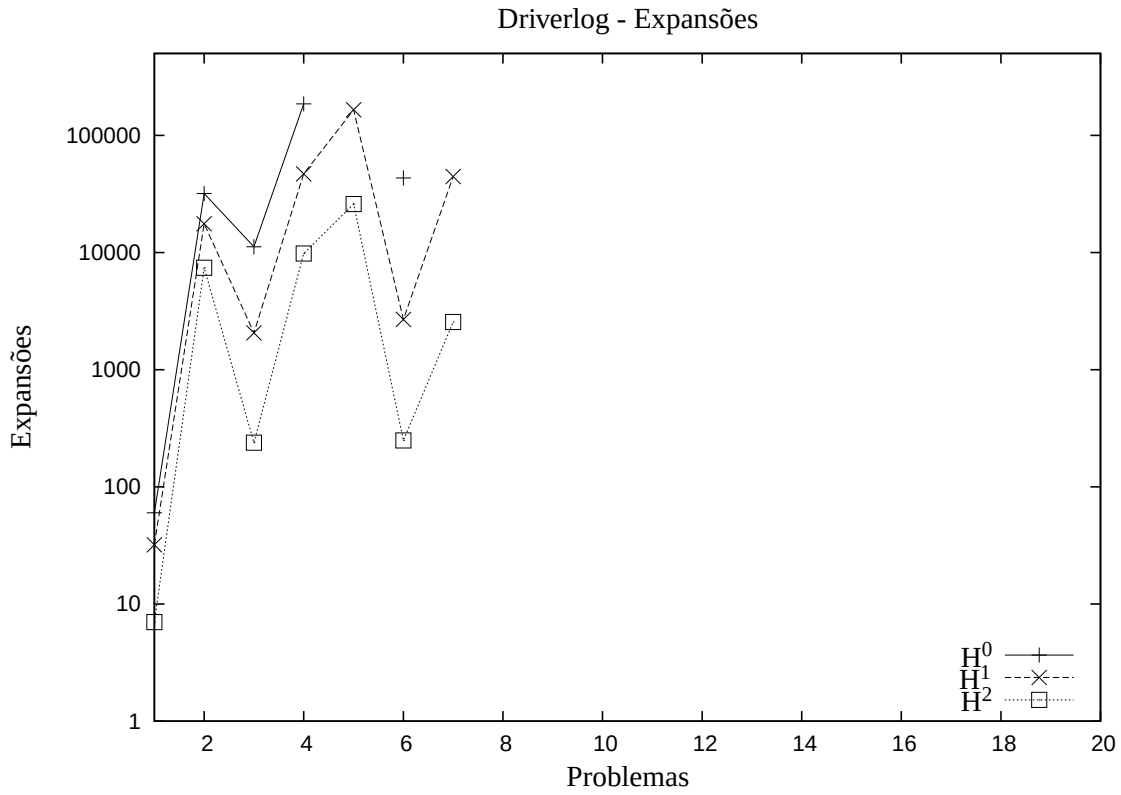


Figura 4.9: Número de expansões do domínio *Driverlog*

165 000 expansões utilizando a heurística H^1. Em contrapartida, o problema 14 do domínio *Blocksworld* gerou 865 000 eventos para realizar 223 000 expansões. Essa diferença aparece também quando compara-se os dados referentes ao número médio de dependências encontradas no vetor de cálculo, mostrado na figura 4.11.

As mudanças do número médio de dependências, apresentadas na figura 4.11, podem parecer pequenas à primeira vista, mas salienta-se que o tamanho dos vetores de cálculo H^1 e H^2 é proporcional ao número de lugares da rede de Petri. Além disso, o número de lugares e transições também infuencia diretamente na velocidade do algoritmo de desdobramento de redes de Petri. Neste caso, justamente os problemas maiores, onde o uso de heurísticas é mais necessário, são os problemas onde o cálculo é mais lento. Mesmo assim, em todos os exemplos das redes de Petri em que a solução ótima não foi encontrada no tempo limite de 2 500 segundos, a heurística H^2 chegou mais perto da solução do que a heurística H^1 ou H^0, conforme mostra a figura 4.12.

A própria utilização do algoritmo A* já faz com que o tempo global de execução seja maior do que em uma busca onde não se objetiva o plano ótimo. O algoritmo A* só realiza o teste de solução no momento em que o evento é retirado da primeira posição da lista de prioridades de de eventos não expandidos. Até o momento em que o plano ótimo é posicionado na primeira posição da lista de prioridades, várias outras soluções diferentes do plano ótimo podem ser

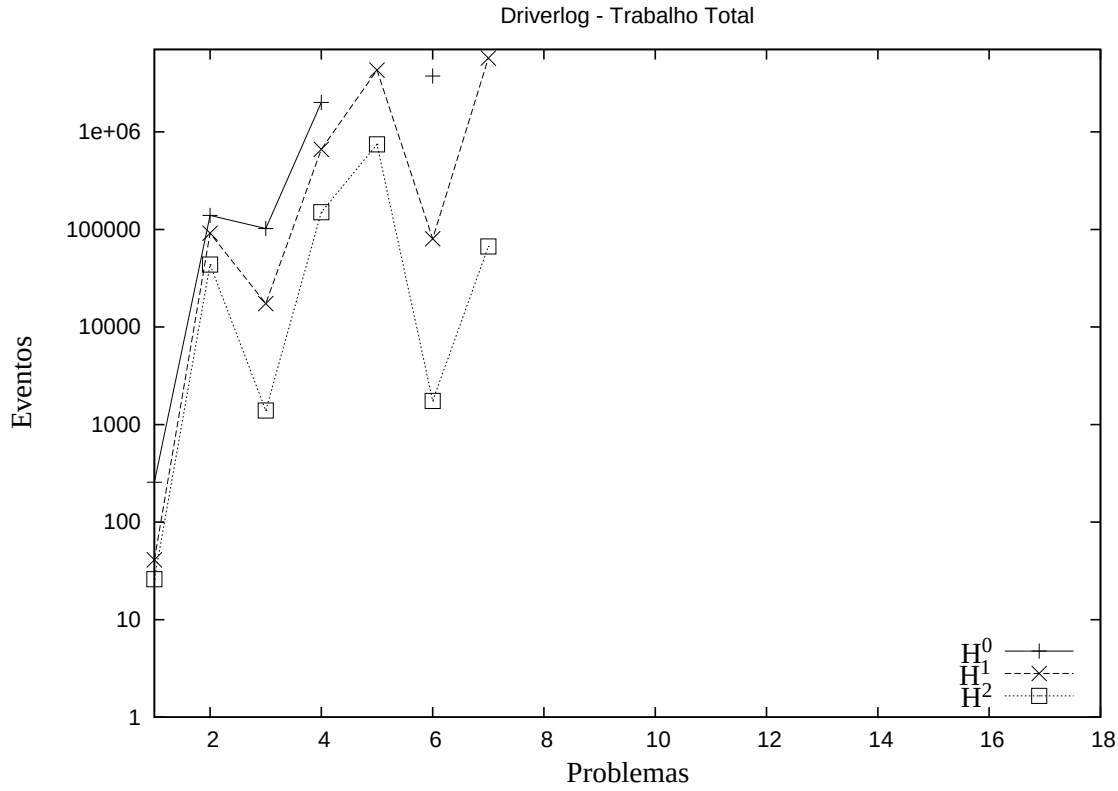


Figura 4.10: Eventos gerados no domínio *Driverlog*

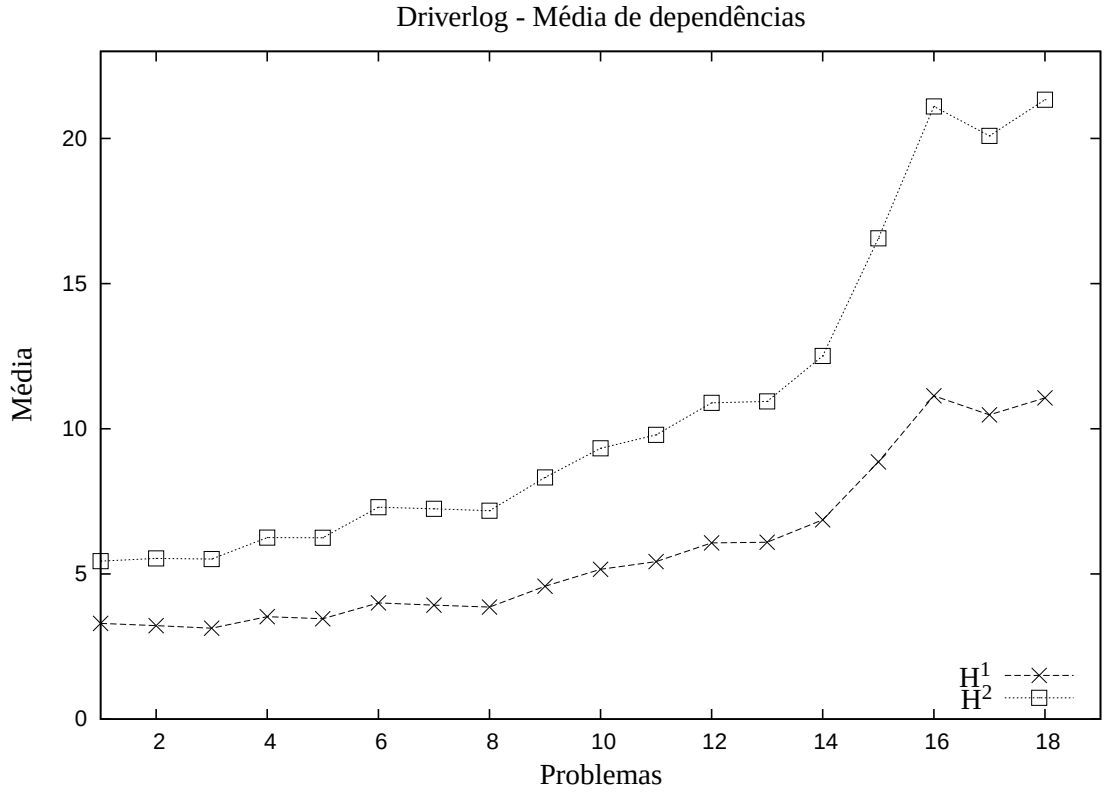


Figura 4.11: Número médio de dependências do domínio *Driverlog*

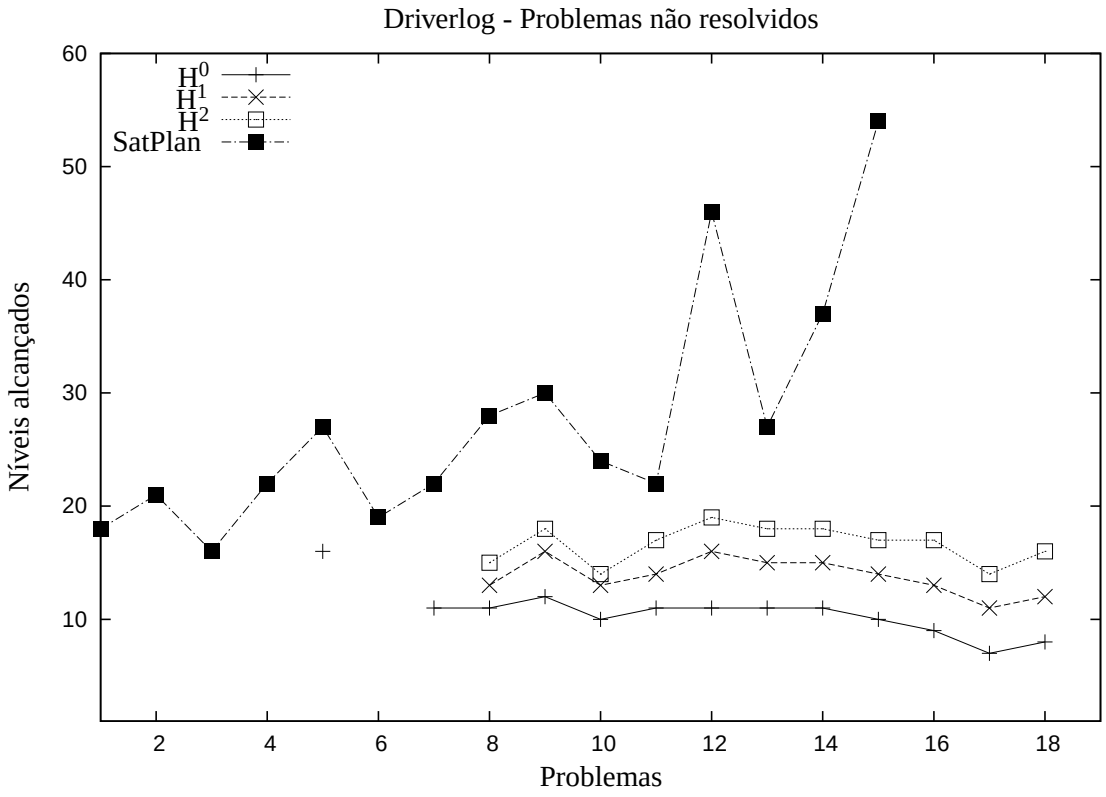


Figura 4.12: Profundidade explorada pelos problemas não-resolvidos no domínio *Driverlog*

encontradas e inseridas na lista. A fim de exemplificar esta característica, figura 4.13 mostra, para o problema número 5, a relação entre o número de expansões e o tempo de execução do *Mole*, utilizando a heurística H^0. Também estão marcados no gráfico as soluções encontradas antes do *Mole* chegar ao plano ótimo.

Mesmo sem a utilização de uma heurística, uma solução foi encontradas em 5.43 segundos de execução. Entretanto, esta primeira solução possui um plano de tamanho 20, quando o plano ótimo deste exemplo possui tamanho igual a 13. Uma vez que a implementação do algoritmo A* pressupõe que o algoritmo só pode ser encerrado quando a primeira posição do vetor de nós a serem expandidos contiver a solução, esta primeira solução é inserida na lista de prioridades de eventos não expandidos, e o *Mole* continua a busca. Ao todo, são encontradas 425 soluções antes da solução de custo igual a 13 se posicionar na primeira posição da lista de prioridades, o que só ocorre aos 4566 segundos. O *Mole* encontrou a primeira solução aos 12.44 segundos e a solução ótima aos 953 segundos quando orientado pela heurística H^1 e encontrou a primeira solução aos 105 segundos e a solução ótima aos 407 segundos quando orientado pela heurística H^2.

O gráfico da figura *driverlog-partial* também demonstra a taxa de expansões variável da execução do *Mole*. As primeiras 1000 expansões foram realizadas em 0.076378 segundos. Fazendo a divisão do número de expansões pelo tempo, chega-se a uma taxa de expansões

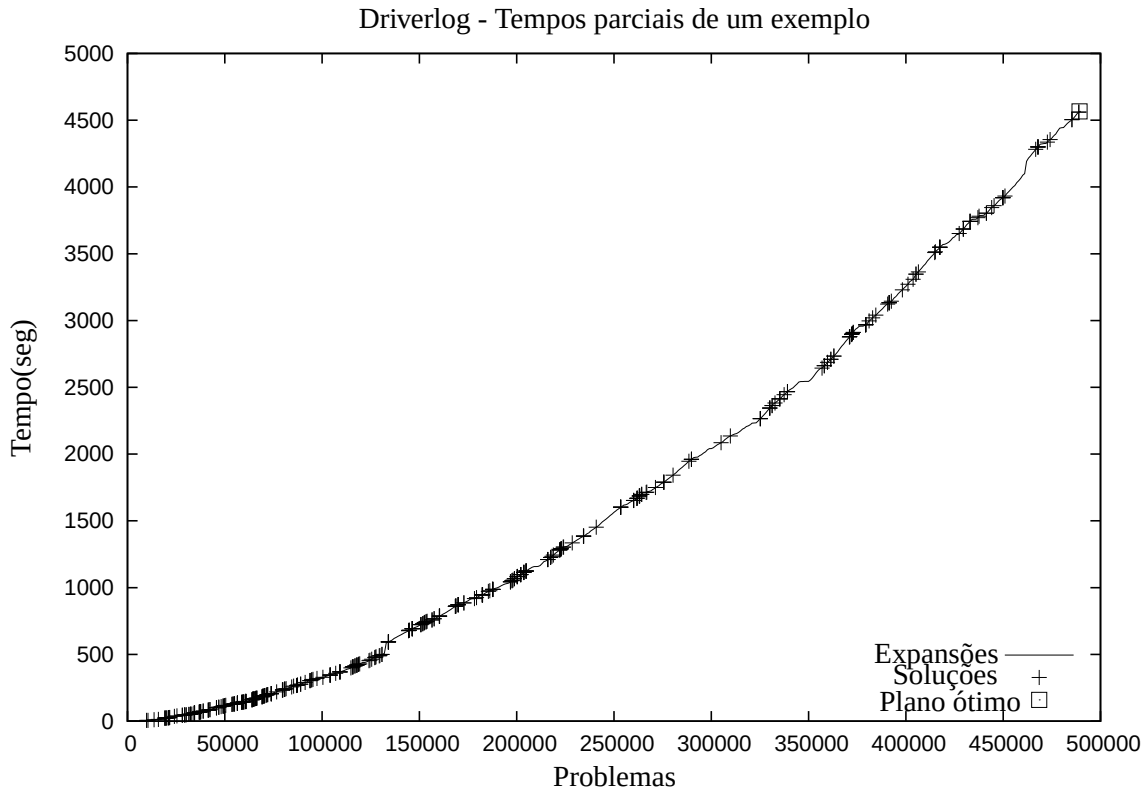


Figura 4.13: Tempos parciais de um exemplo do domínio *Driverlog*

de 13092.77 expansões por segundo. Esta taxa vai caindo progressivamente, até o momento em que o desdobramento é encerrado, o que ocorre com a realização de 489234 expansões aos 4566.208857 segundos, o que retorna uma taxa de expansões de 107,14 expansões por segundo.

4.2.3 Logistics

Da mesma forma que o domínio *Driverlog*, o domínio *Logistics* representa um problema de logística de pacotes. Existem pacotes que devem ser transportados para diferentes lugares. Os pacotes são transportados por caminhões, quando as localidades são na mesma cidade, ou por aviões, quando as localidades estão em cidades diferentes, sendo que os aviões só se movimentam entre aeroportos.

A figura 4.14 apresenta o número de lugares e transições das redes de Petri geradas a partir dos problemas de planejamento do domínio *Logistics*, e a figura 4.15 apresenta os tempos de execução.

Conforme mostra a figura 4.14, os problemas do domínio *Logistics* podem ser divididos em quatro grupos de complexidade, no que se refere ao número de lugares e transições. Todos os problemas que foram solucionados no tempo limite de 2500 segundos pertencem ao primeiro grupo, ou seja, as redes de Petri com 48 lugares e 85 transições. O segundo grupo engloba

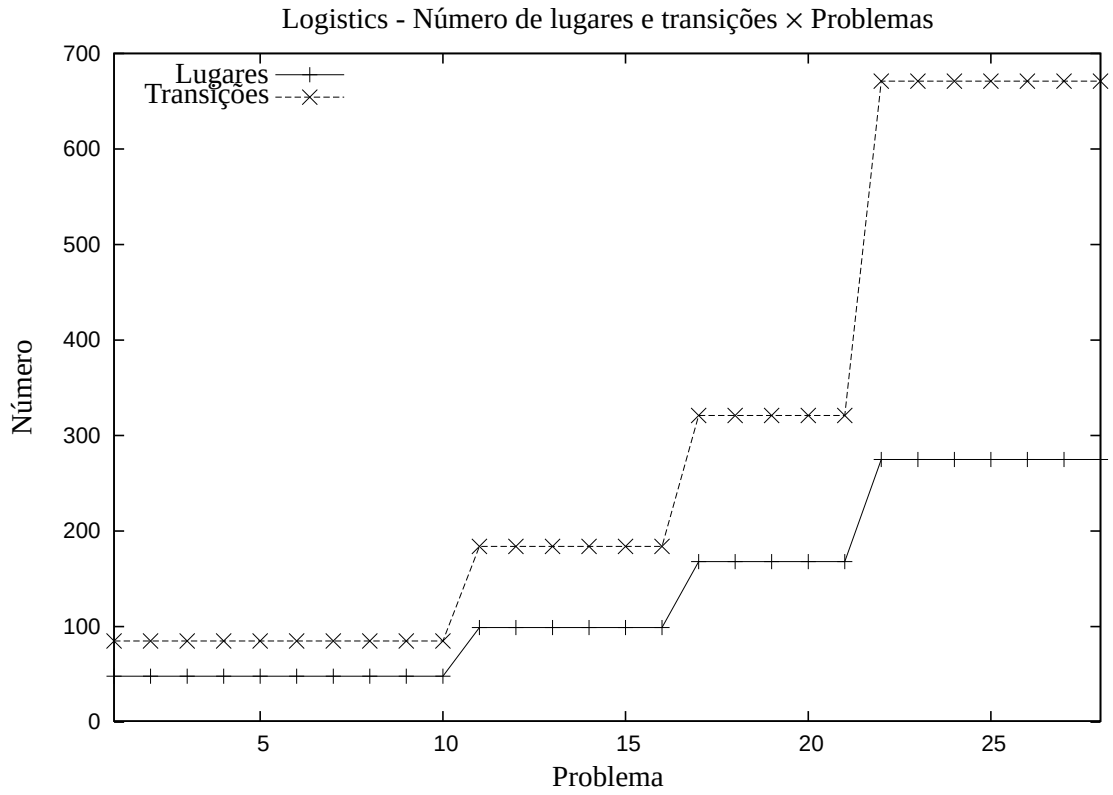


Figura 4.14: Número de Lugares e Transições das redes do domínio *Logistics*

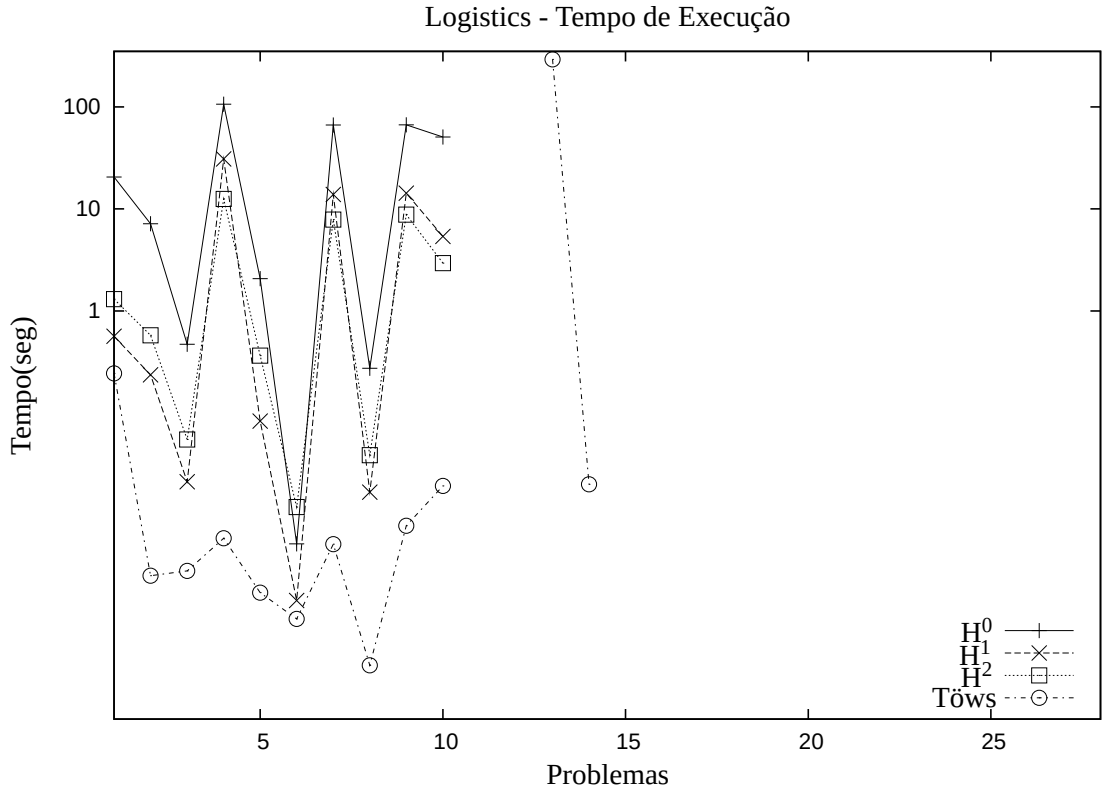


Figura 4.15: Tempo de execução do domínio *Logistics*

redes com o dobro do tamanho do primeiro grupo (99 lugares e 184 transições). Dentro deste primeiro grupo, o maior tempo de execução obtido foi no problema número 4, onde o *Mole* guiado pela heurística H^0 encontrou uma solução aos 106 segundos. A variação do tempo de execução destes problemas esteve diretamente ligada ao tamanho do plano ótimo de cada problema, conforme pode ser observado na figura 4.15. Da mesma forma como ocorreu nos domínios *Driverlog* e *Blocksworld*, nos problemas em que a solução ótima foi encontrada em um tempo inferior a 1 segundo, a heurística H^2 teve desempenho inferior à heurística H^1.

No caso dos problemas do domínio *Logistics*, o *SatPlan* não conseguiu nem interpretar a sintaxe dos problemas em *PDDL*, sendo esta a razão do gráfico da figura 4.15 não apresentar os dados referentes ao *SatPlan*. No caso da implementação de Töws, o desempenho de tempo foi inferior em todos os casos. Percebe-se também no gráfico que o aumento da complexidade dos problemas do domínio *Logistics* também afetou no desempenho da heurística de Töws, pois foram solucionados apenas dois problemas a mais do que com as heurísticas H^1 e H^2.

Em relação ao número de expansões, a heurística H^2 realizou menos expansões do que a heurística H^1 em todos os casos, conforme demonstra a figura 4.16. O número total de eventos gerados pelos problemas do domínio *Logistics* é apresentado na figura 4.17 e o número médio de dependências é demonstrado na figura 4.18.

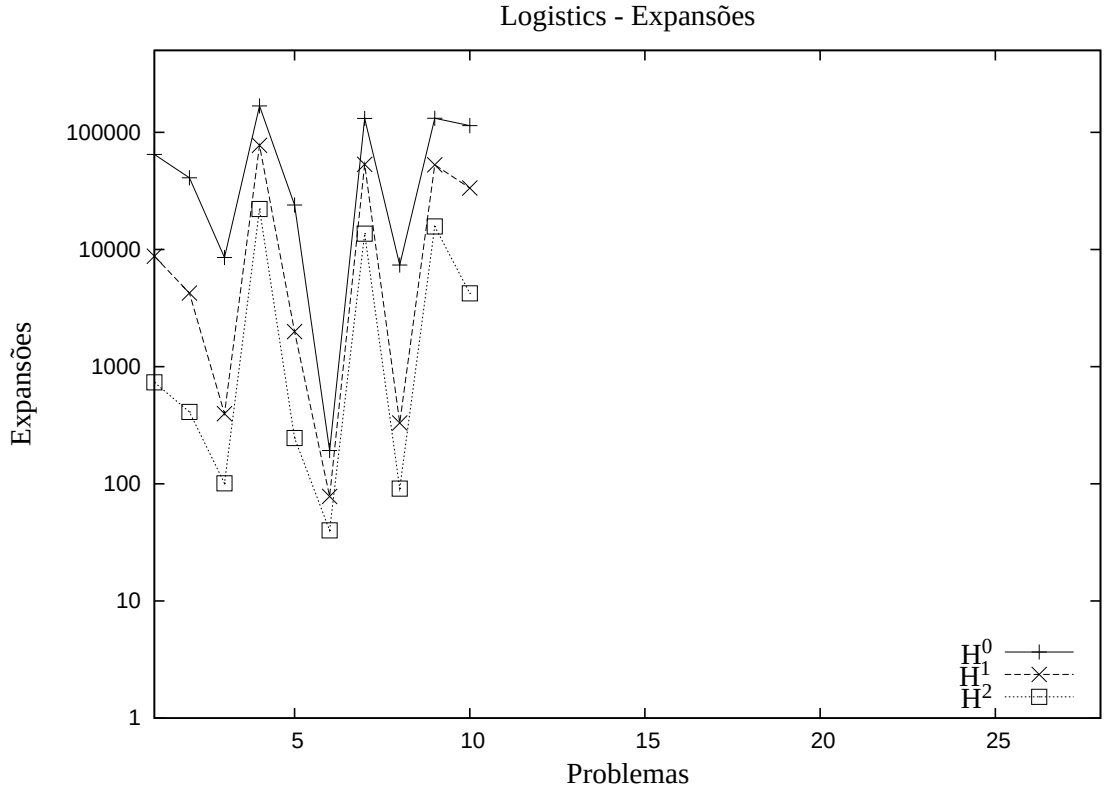


Figura 4.16: Expansões do domínio *Logistics*

O número médio de dependências do domínio *Logistics* não variou muito ao longo dos

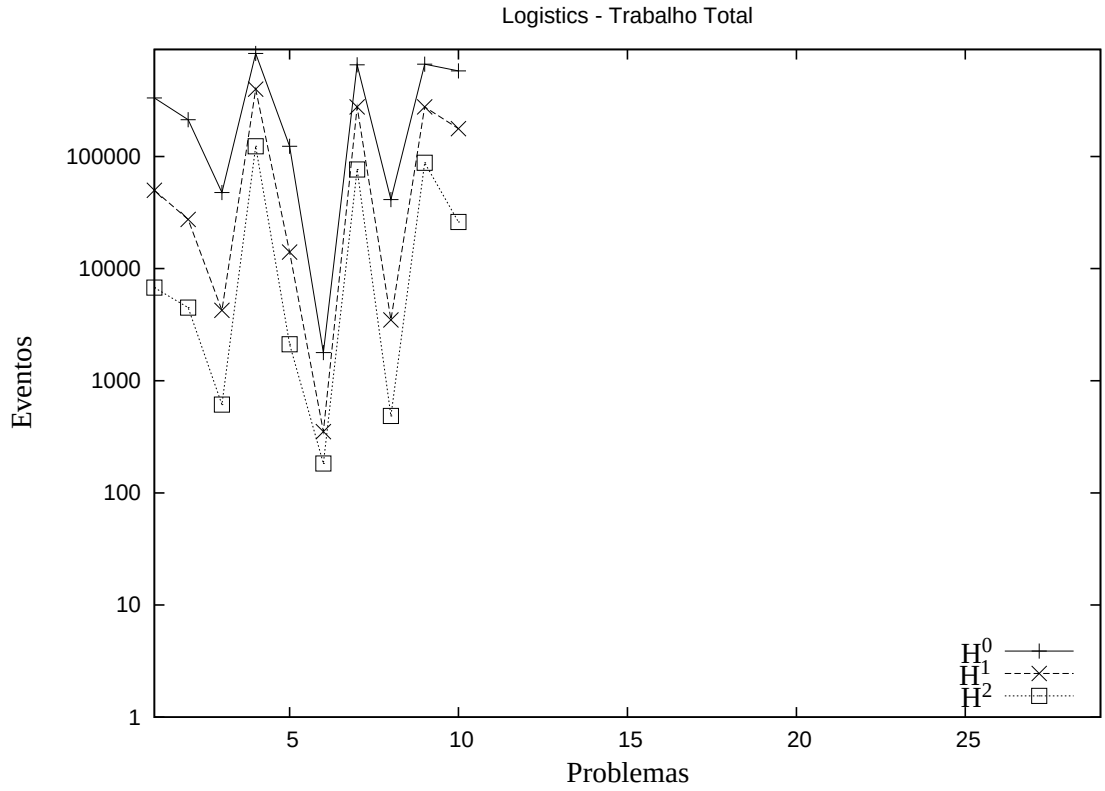


Figura 4.17: Eventos gerados no domínio *Logistics*

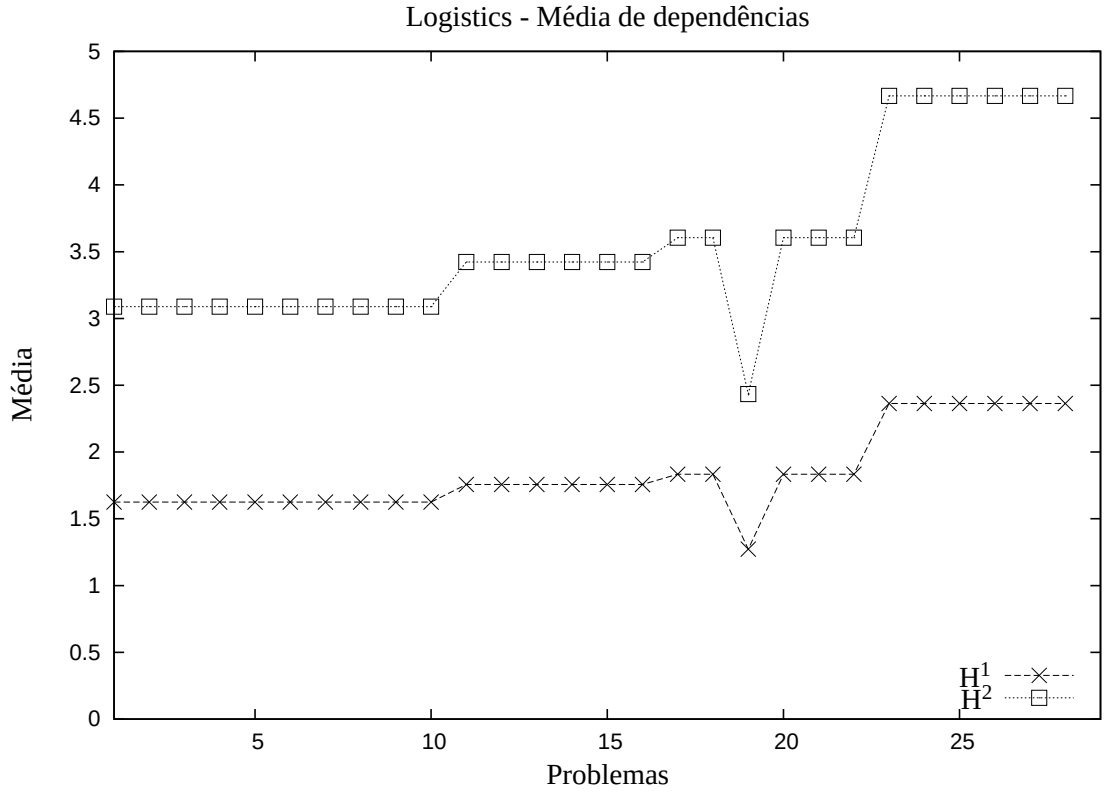


Figura 4.18: Número médio de dependências do domínio *Logistics*

problemas. O percentual não é igual para todos os problemas da mesma classe porque referem-se ao vetor de cálculo depois de eliminados os subconjuntos inalcançáveis, processo descrito na seção 3.2.2.

A figura 4.19 demonstra a profundidade explorada pelos problemas não resolvidos. Como os dados do *SatPlan* não estão disponíveis, o gráfico está relacionando apenas o tamanho do plano ótimo nos problemas que foram solucionados no tempo limite de 2 500 segundos por alguma das heurísticas utilizadas.

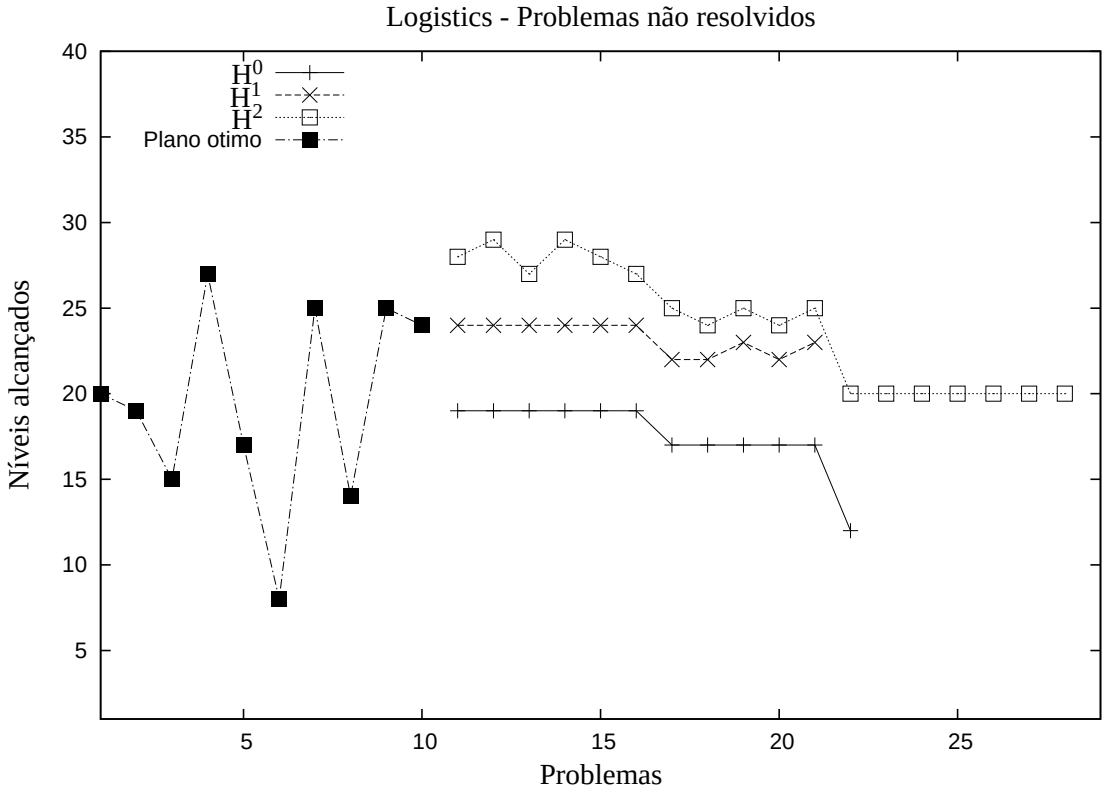


Figura 4.19: Profundidade explorada pelos problemas não-resolvidos no domínio *Logistics*

Ainda que para este domínio não seja possível conhecer de antemão a profundidade do plano ótimo, percebe-se que, em todos os casos em que o plano ótimo não foi obtido no tempo limite de 2 500 segundos, a heurística H^2 atingiu uma profundidade maior do que a heurística H^1, e esta atingiu uma profundidade maior do que a busca não-orientada.

Uma característica própria do domínio *Logistics* é que o número de eventos gerados para cada expansão nos problemas em que não se encontrou uma solução no tempo limite de 2 500 segundos aumentou consideralvelmente, quando comparados com o dado dos problemas resolvidos. Utilizando a heurística H^0, o problema 11 realizou 576 000 expansões, utilizando para isso 5 207 000 eventos. O problema 18 realizou 519 000 expansões, utilizando 7 718 000 eventos. Já o problema 23 realizou 62 000 expansões, utilizando para isso 25 336 000 eventos. Na categoria do problema 23, que engloba os problemas com 275 lugares e 671 transições, os da-

dos da figura 4.19 estão incompletos para as heurísticas H^0 e H^1 porque a alocação de memória superou os 20 Gb disponíveis para os testes antes da busca ser encerrada por exceder o tempo limite. A utilização da heurística H^2 não excedeu o limite de memória porque a taxa de expansões da heurística H^2 é mais lenta do que a taxa das heurísticas H^1 ou H^0, o que reflete na utilização de menos expansões e consequentemente menor uso de memória.

4.2.4 Zenotravel

O domínio *Zenotravel* é um domínio que define o transporte de passageiros entre localidades, com velocidades definidas independentemente para cada avião. A figura 4.20 apresenta o número de lugares e transições do domínio e a figura 4.21 apresenta os tempos de execução dos problemas deste domínio.

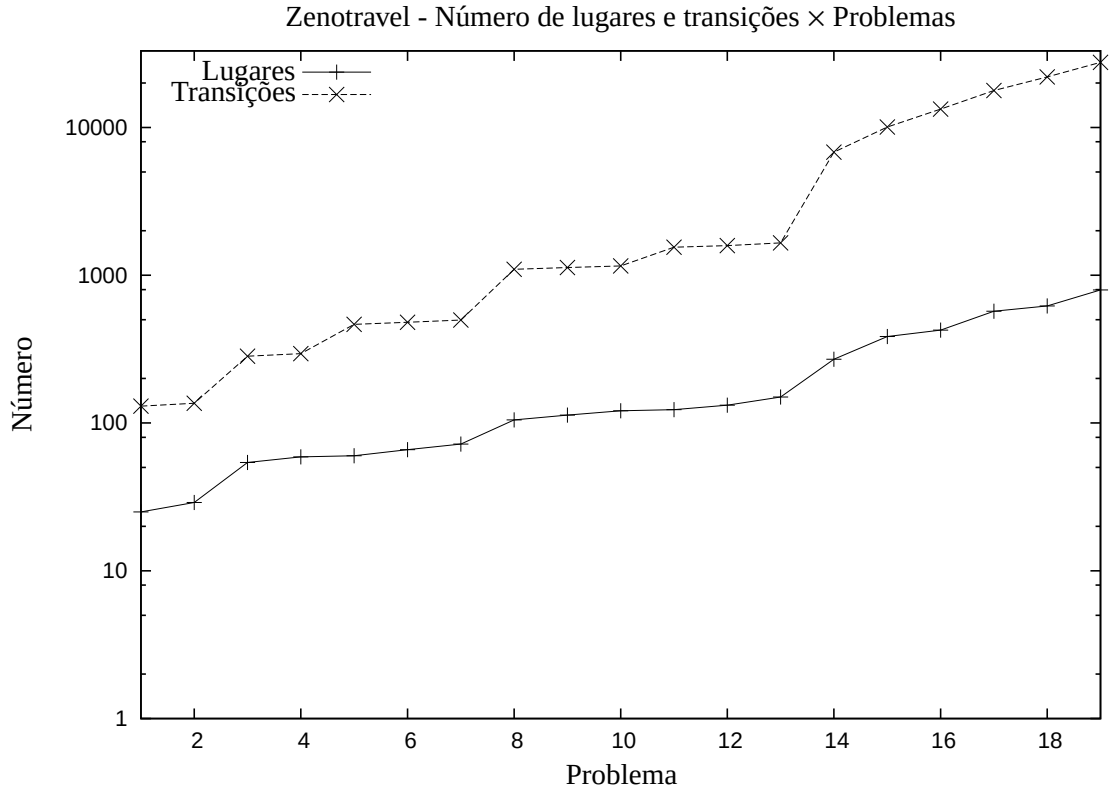


Figura 4.20: Número de Lugares e Transições das redes do domínio *Zenotravel*

De todos os domínios que fizeram parte do conjunto de testes, o domínio *Zenotravel* possui as redes de Petri com a maior diferença entre o número de lugares e o número de transições. Enquanto que no domínio *Driverlog* a relação entre o número de lugares e transições nunca passou de 10 transições para cada lugar, a menor rede de Petri gerada a partir do domínio *Zenotravel* possui 25 lugares e 130 transições, resultando em uma relação de 5 transições para cada lugar e a maior rede possui 855 lugares e 32 781 transições, resultando em uma relação de 38 transições para cada lugar. A variação do tempo de execução entre os problemas teve

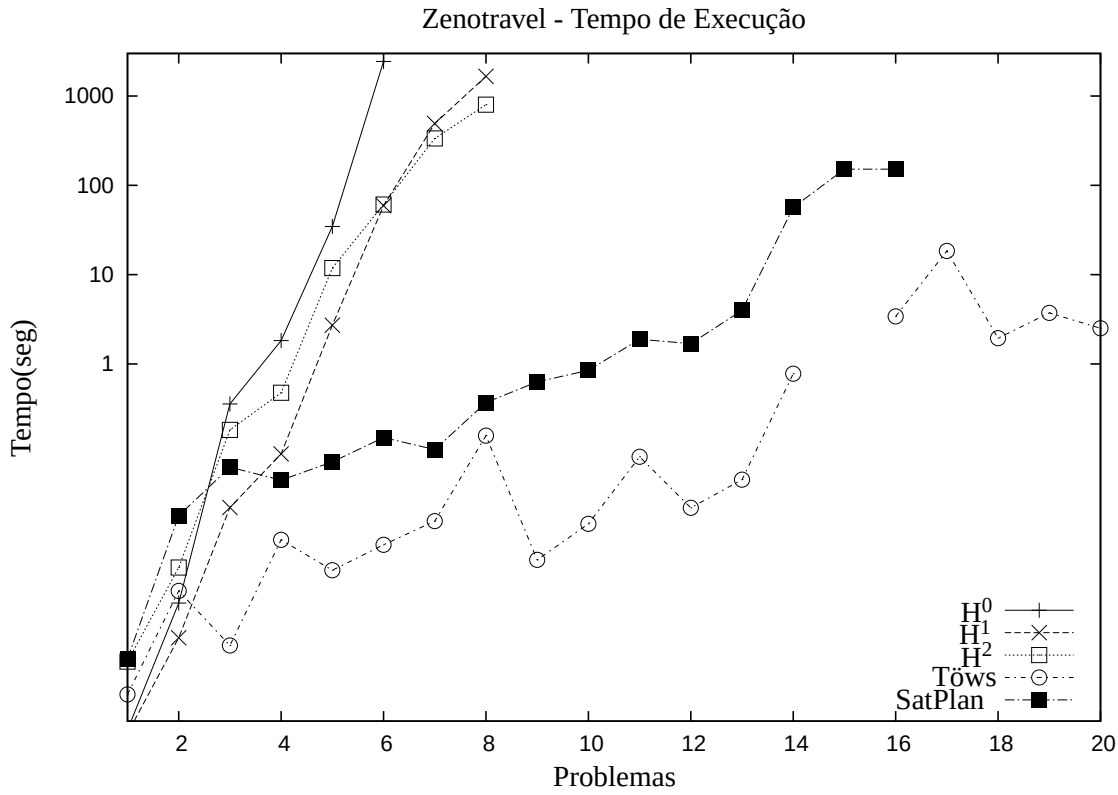


Figura 4.21: Tempo de execução do domínio *Zenotravel*

uma relação direta com o aumento da complexidade dos problemas. No caso do desempenho de tempo das heurísticas, o mesmo padrão encontrado nos domínios anteriores manteve-se no domínio *Zenotravel*, pois a heurística H^2 teve um desempenho inferior à heurística H^1 nos problemas de número 1 a 5, mas passou a melhorar à medida em que a complexidade do problema aumentou. No exemplo 2, a heurística H^1 teve um desempenho superior ao *SatPlan*, à heurística H^2 e à heurística de Töws. Nos outros problemas, manteve-se o mesmo padrão encontrado no domínio *Blocksword*, onde as buscas orientadas deste trabalho apresentam resultados melhores do que o *SatPlan* apenas nos problemas de pequeno porte.

A figura 4.22 apresenta o número de expansões realizadas pelo *Mole* no domínio *Zenotravel*. Da mesma maneira em que ocorreu nos demais domínios, o número de expansões realizadas pela heurística H^2 foi menor do que o número de expansões da heurística H^1, em todos os casos.

Como já foi comentado, o domínio *Zenotravel* caracteriza-se por possuir a maior diferença entre o número de transições e lugares, comparado com os outros domínios analisados. Esta diferença também se reflete no trabalho total realizado pela busca, apresentado na figura 4.23 e pela complexidade do vetor de cálculo, apresentada na figura 4.24.

Uma análise do trabalho efetivo realizado nos problemas do domínio *Zenotravel* demonstram uma grande diferença entre o número de expansões realizadas e o número de eventos gerados. Sem dúvida, o número de transições dos problemas deste domínio contribuiu para

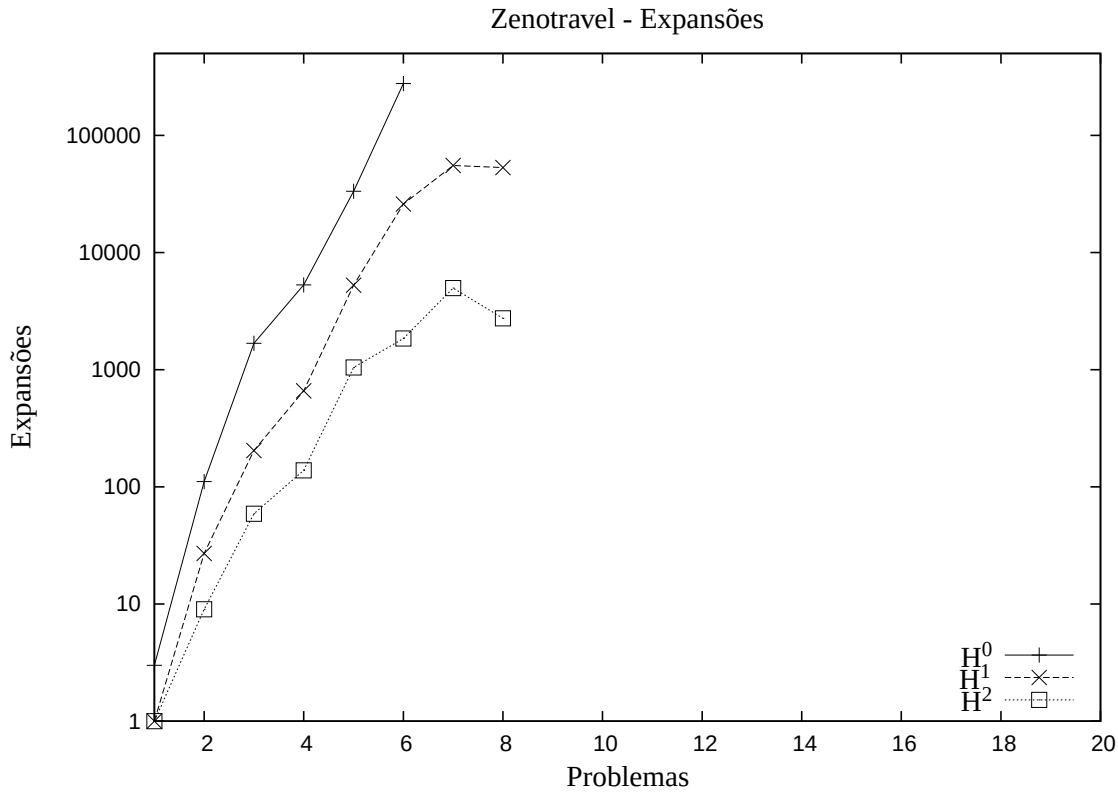


Figura 4.22: Expansões do domínio *Zenotravel*

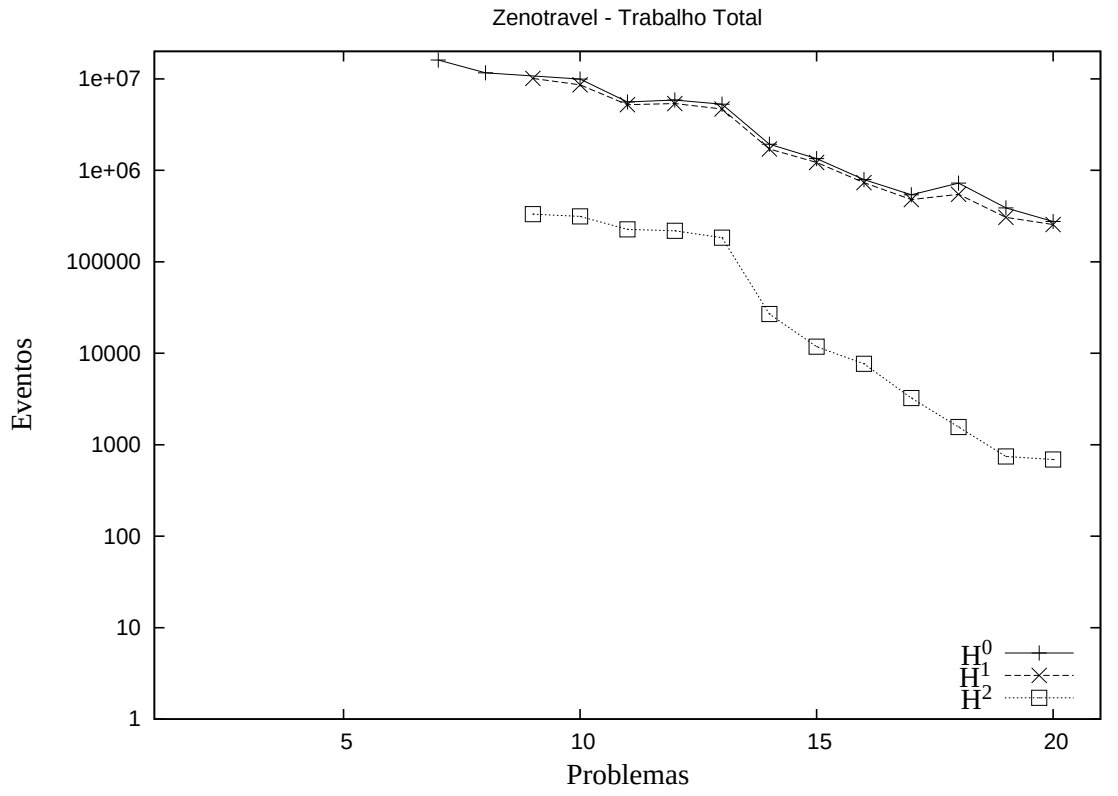


Figura 4.23: Eventos gerados no domínio *Zenotravel*

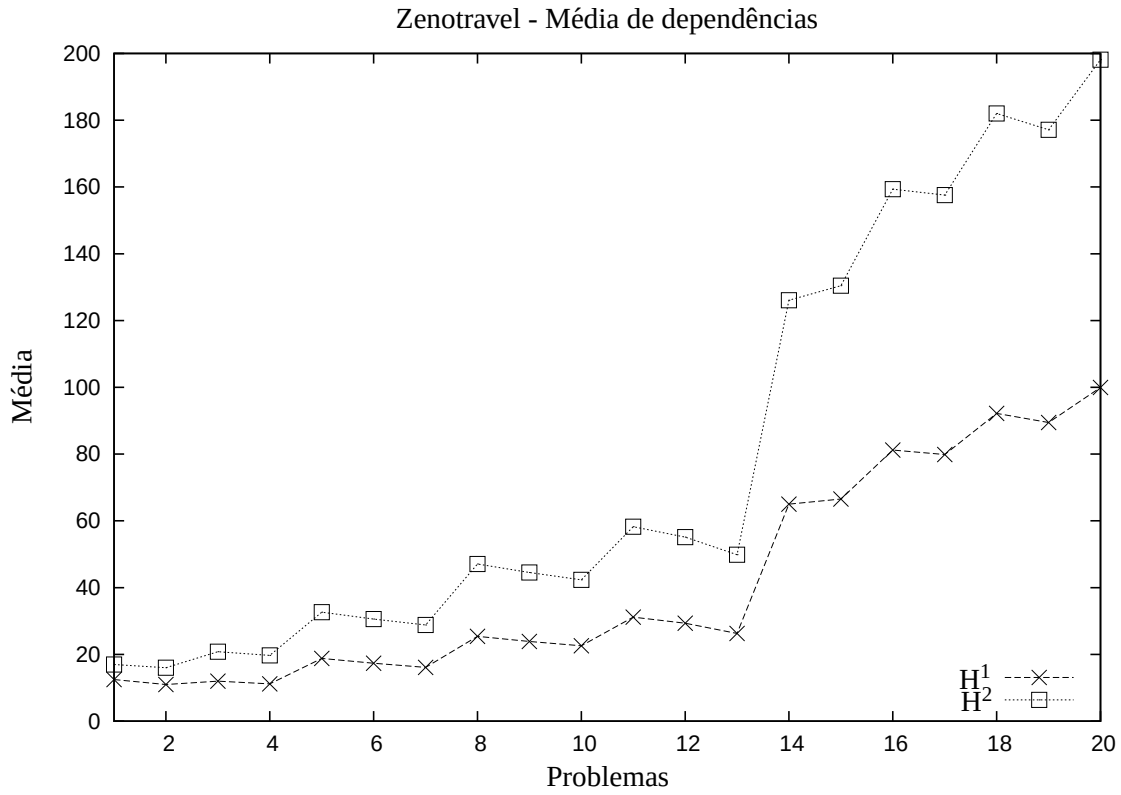


Figura 4.24: Número médio de dependências do domínio *Zenotravel*

que, neste exemplo, o volume de trabalho efetivo seja tão grande quando comparado ao número de expansões. À medida em que a complexidade dos problemas cresce, a taxa de expansões do algoritmo de desdobramento cai, conforme mostra a figura 4.23. Por esta razão, a profundidade atingida pelo cálculo também ficou prejudicada, como demonstra a figura 4.25.

Na figura 4.25, mostra-se que a complexidade dos problemas fez com que a profundidade alcançada diminuísse bastante. Nos problemas de número 16 a 20, o próprio *SatPlan* não conseguiu encontrar a solução. Nos problemas de maior complexidade, a complexidade do vetor de cálculo tornou extremamente ineficiente a busca através do algoritmo A*. Nos problemas de número 17 a 20, a heurística H^2 conseguiu, no máximo, a mesma eficiência obtida pela heurística H^1, que teve uma eficiência um pouco acima do obtido pela heurística H^0.

4.2.5 Rovers

O domínio *Rovers* é formado por veículos de exploração autômatos, que devem navegar na superfície de um planeta a procura de amostras de solo. As amostras devem ser coletadas e depositadas em um módulo de pouso.

A figura 4.26 apresenta o tamanho dos problemas do domínio *Rovers* e a figura 4.27 apresenta o tempo de execução dos referidos problemas.

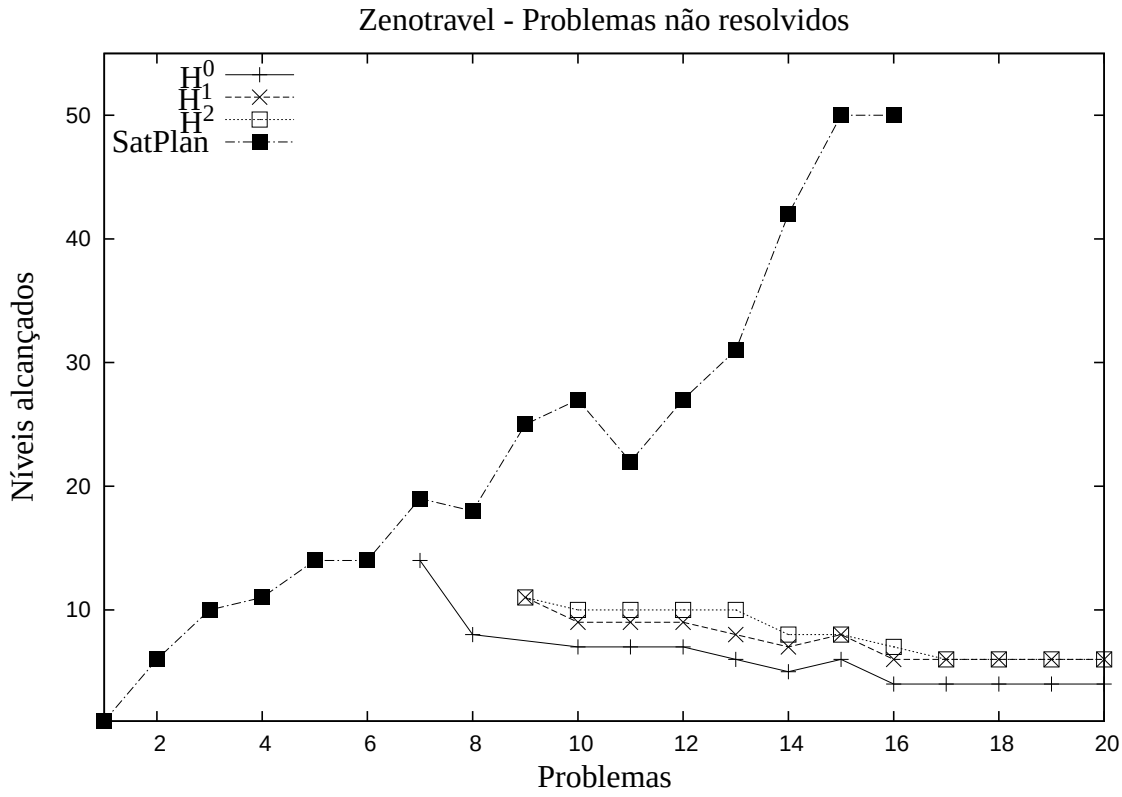


Figura 4.25: Profundidade explorada pelos problemas não-resolvidos no domínio *Zenotravel*

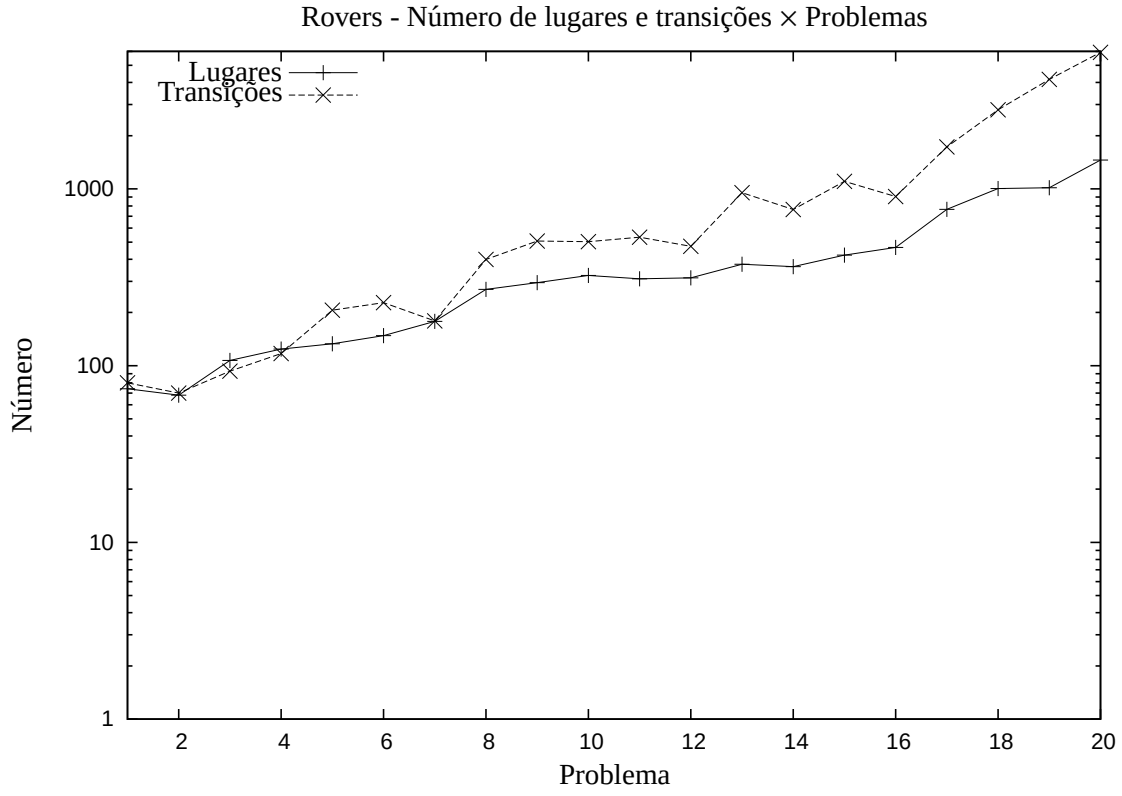


Figura 4.26: Número de Lugares e Transições das redes do domínio *Rovers*

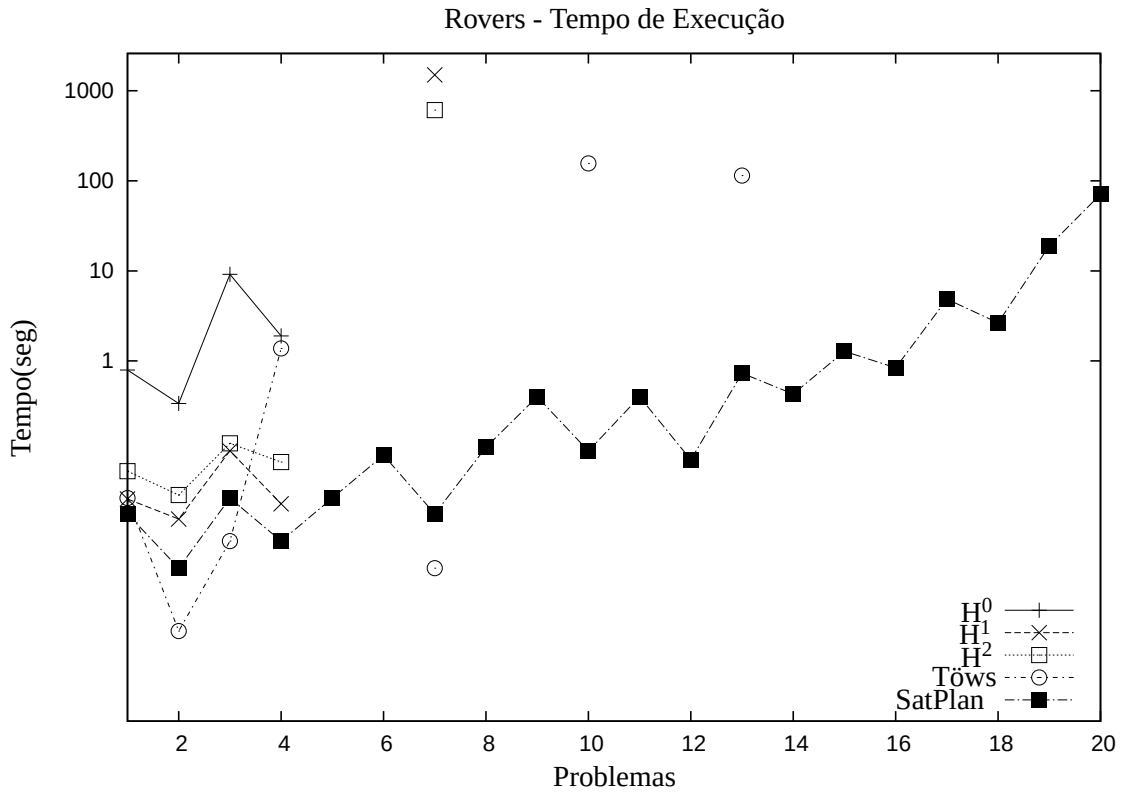


Figura 4.27: Tempo de execução do domínio *Rovers*

O crescimento da complexidade dos problemas possui uma diferença singela em relação ao crescimento da complexidade dos problemas de outros domínios: Enquanto que, nos outros domínios, o número de transições sempre supera o número de lugares, o que reflete diretamente na complexidade do vetor de cálculo, neste domínio o número de lugares chega a ser maior do que o número de transições em algumas instâncias do problema. Este detalhe influencia na complexidade do vetor de cálculo, e consequentemente no desempenho do algoritmo.

O problema de número 7 é um exemplo de como a diferença do número de lugares e transições afetou o tempo de busca, pois este problema possui 178 lugares e 179 transições. Mesmo tendo mais lugares do que os problemas 5 e 6, o problema 7 pode ser solucionado pelo *Mole* em um tempo menor do que 2 500 segundos. Os tempos dos outros problemas que foram solucionados seguiram o mesmo padrão encontrado em problemas dos domínios anteriores, ou seja, a heurística H^2 foi menos efetiva do que a heurística H^1, mas no problema 7, de maior porte, o resultado do tempo de execução da heurística H^2 foi mais eficiente. Entretanto, mesmo que a diferença entre o número de lugares e transições não siga um padrão constante, ainda assim o número de lugares das redes de Petri cresce de maneira mais rápida do que foi verificado em outros domínios.

Em virtude da complexidade dos problemas do domínio *Rovers*, o tempo de execução de todos os problemas solucionados no tempo limite de 2 500 segundos foi maior do que o tempo

do domínio *SatPlan* e *Rovers*.

A comparação do número de expansões realizadas das heurísticas não apresentou nenhuma anomalia em relação aos resultados de domínios anteriores, conforme mostra a figura 4.28. A heurística H^2 teve um desempenho melhor que a heurística H^1, que por sua vez teve um desempenho melhor do que a heurística H^0.

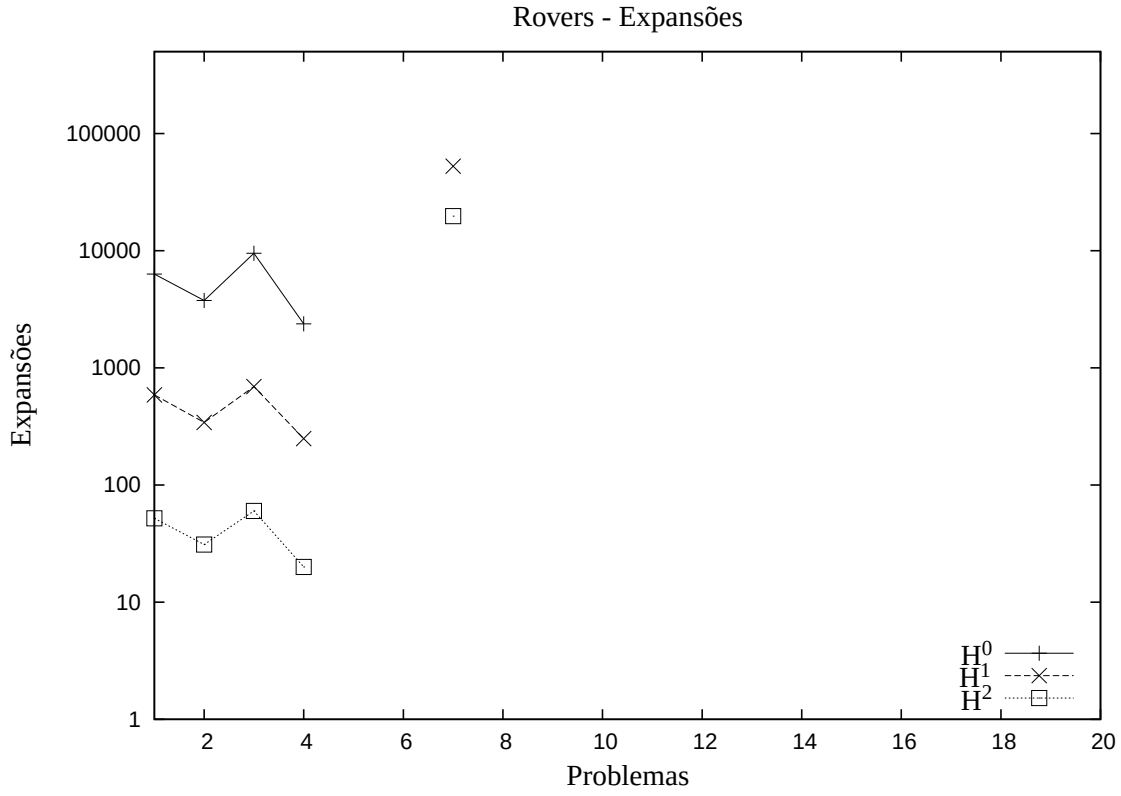


Figura 4.28: Expansões do domínio *Rovers*

O trabalho total realizado é apresentado na figura 4.29 e a média de dependências por elemento do vetor de cálculo é apresentado na figura 4.30. A complexidade dos vetores de cálculo acompanhou a relação entre o número de lugares e transições dos problemas, mostrado na figura 4.26. O problema 7, por exemplo, possui 178 lugares e 179 transições e teve um número médio de dependências menor do que o problema 6, com 148 lugares e 247 transições.

A figura 4.31 apresenta os dados referentes à profundidade alcançada pelos problemas não resolvidos. O maior plano encontrado no domínio *Rovers* foi o plano do problema 7, com 20 elementos. O problema 5 não teve a solução encontrada. Entretanto, a heurística H^2 chegou bem próxima da solução, pois o plano possui tamanho igual a 22 elementos e a heurística H^2 chegou à profundidade 20. O domínio *Rovers* possui o maior plano encontrado em todos os domínios, que é o plano do problema 20 com 95 elementos.

O aumento da complexidade dos problemas resultou na diminuição da profundidade alcançada, da mesma forma como ocorreu em outros domínios.

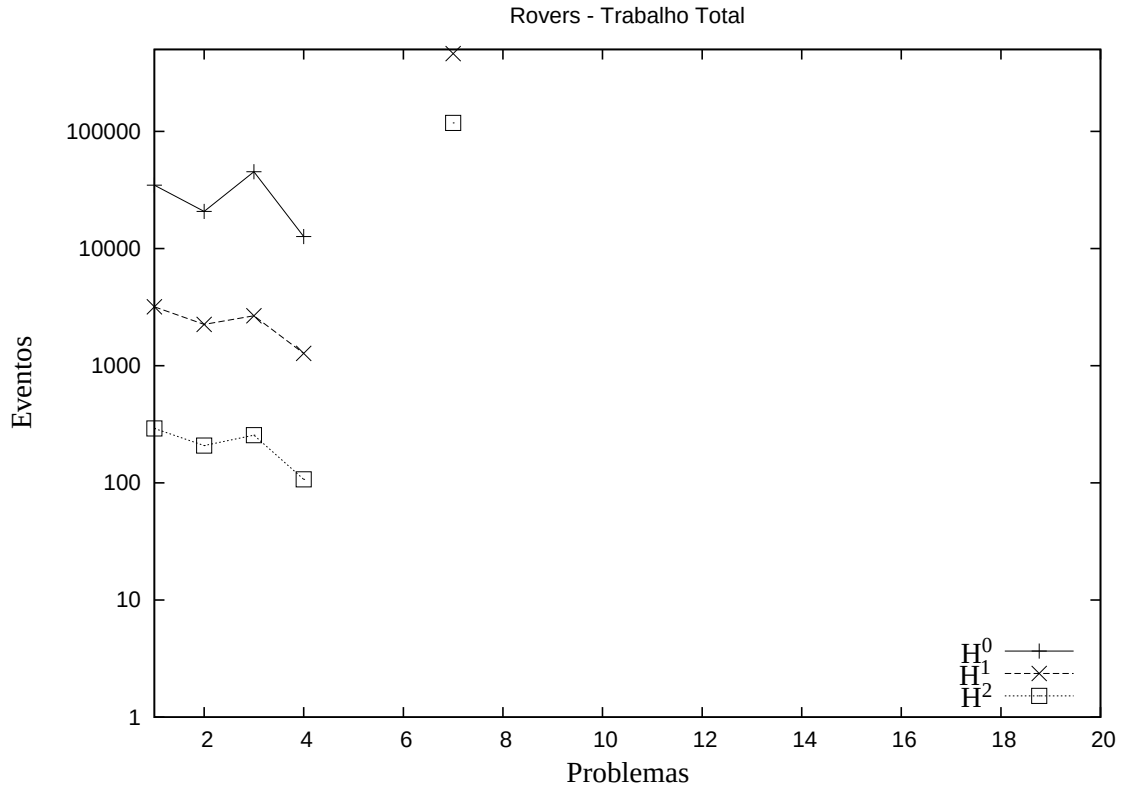


Figura 4.29: Eventos gerados no domínio *Rovers*

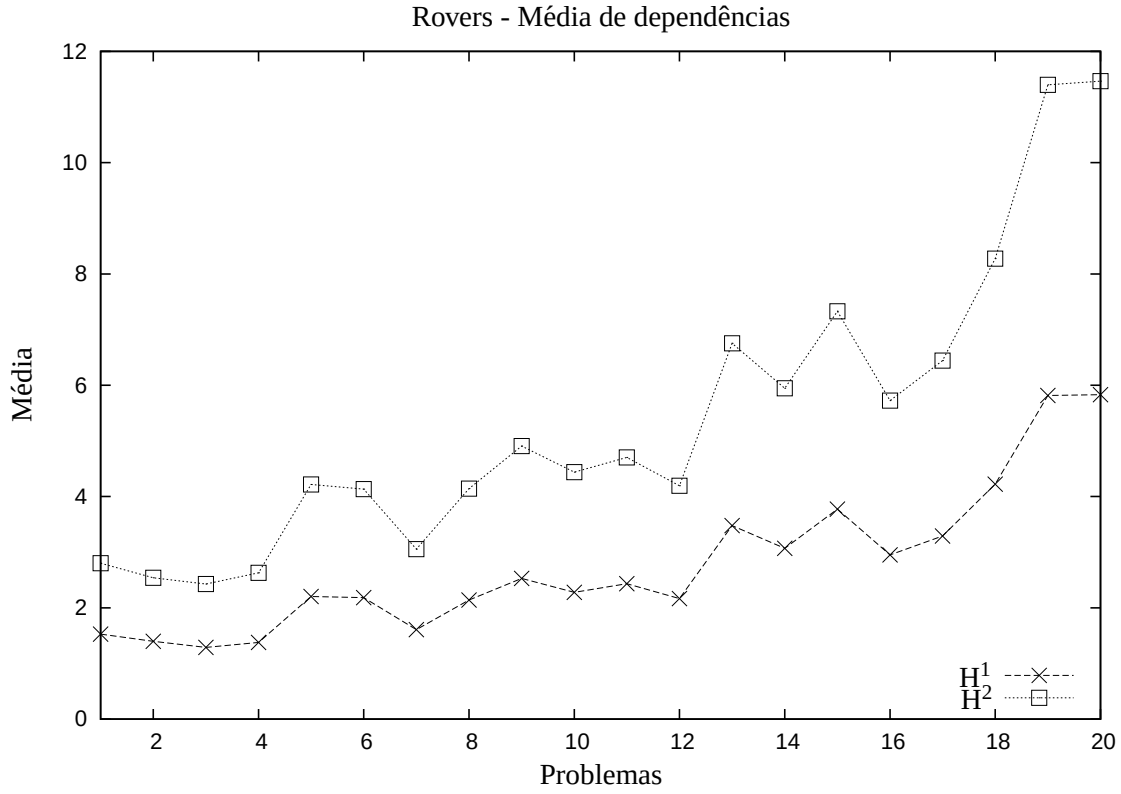


Figura 4.30: Número médio de dependências do domínio *Rovers*

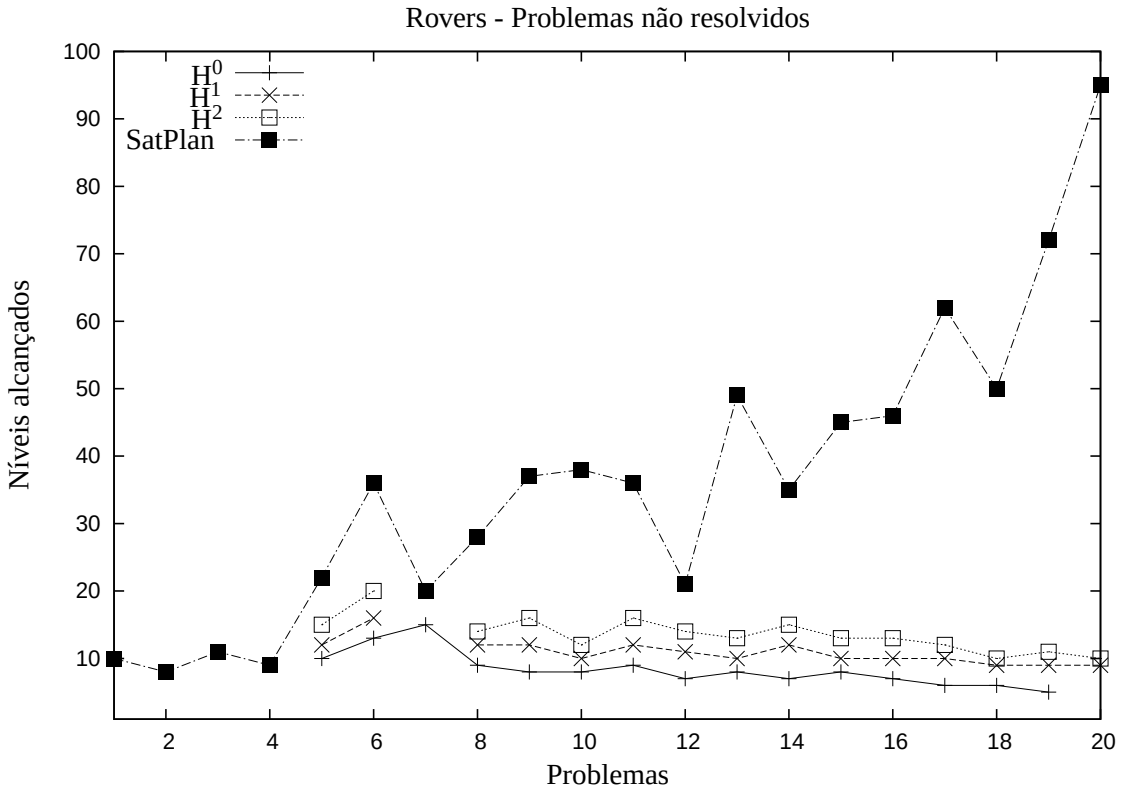


Figura 4.31: Profundidade explorada pelos problemas não-resolvidos no domínio *Rovers*

4.2.6 Satellite

Satélites espaciais são equipados com instrumentos a bordo para registrar diferentes fenômenos. O registo do fenômeno exige que o satélite selecione o instrumento, calibre-o e então tire a foto. Apenas um instrumento pode estar calibrado por vez.

A figura 4.32 apresenta o número de lugares e transições do domínio *Satellite* e a figura 4.33 apresenta os tempos de execução dos problemas deste domínio.

Dentre todos os domínios analisados, o domínio *Satellite* foi aquele que obteve menos problemas solucionados em relação ao número total de problemas. De um universo de 20 problemas, apenas 3 foram solucionados. A utilização da heurística H^0 foi eficaz apenas no problema de número 1. Neste problema, a heurística H^1 teve desempenho superior ao desempenho da heurística H^2. Nos problemas 2 e 3, a heurística H^2 teve desempenho de tempo superior ao desempenho da heurística H^1.

Pode-se dizer que o domínio *Satellite* possui os problemas com o maior tamanho médio dentre os domínios analisados. Ainda que o domínio *Zenotravel* possua a maior diferença entre o número de lugares e transições, e em outros domínios existem problemas maiores do que os problemas do domínio *Satellite*, seus domínios possuem problemas de pequeno porte, cuja complexidade vai crescendo até o ponto em que nenhuma das heurísticas utilizadas se torna

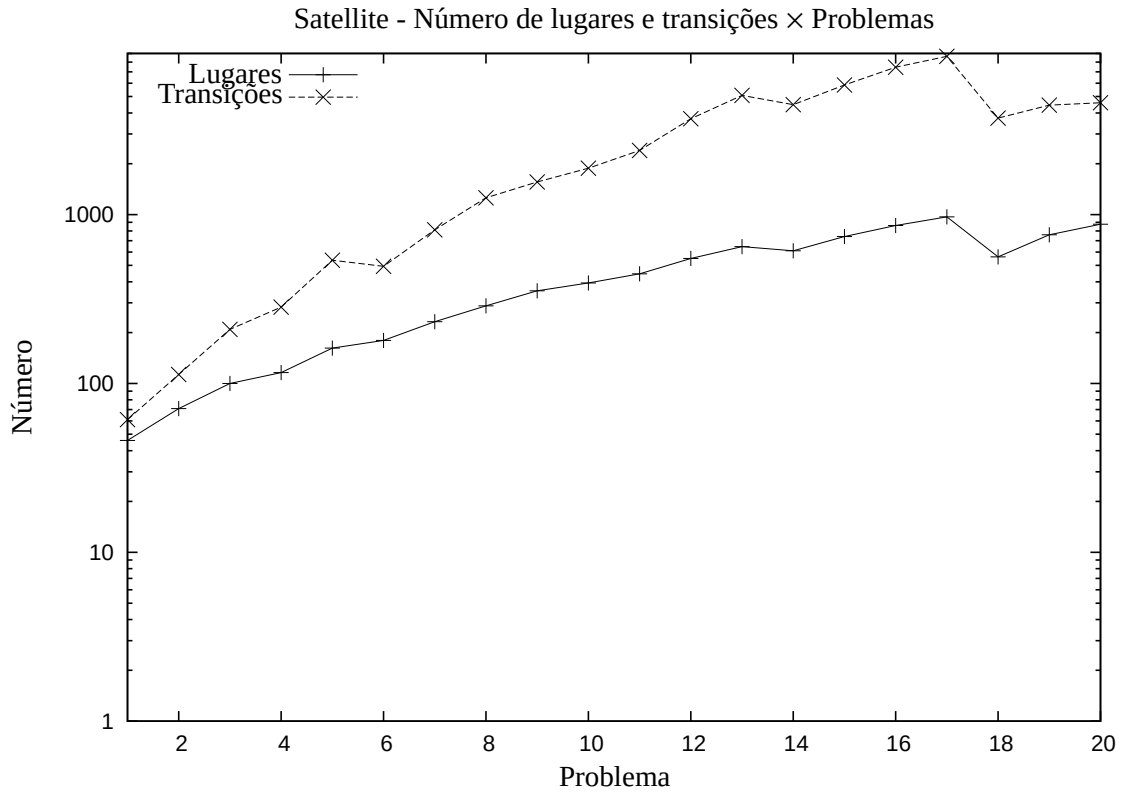


Figura 4.32: Número de Lugares e Transições das redes do domínio *Satellite*

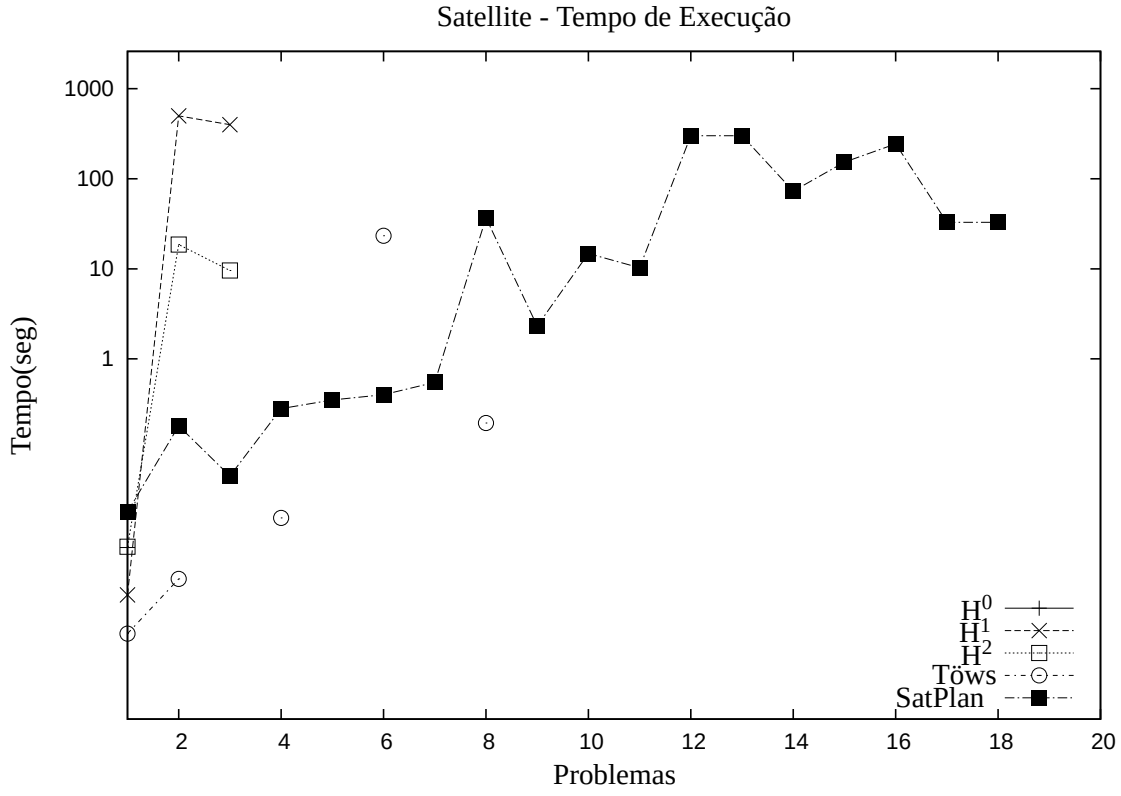


Figura 4.33: Tempo de execução do domínio *Satellite*

eficaz na tarefa de encontrar uma solução no tempo limite. O ritmo em que a complexidade dos problemas do domínio *Satellite* cresce é maior do que o ritmo de outros domínios. O primeiro problema possui 46 lugares e 61 transições. O segundo problema possui 71 lugares e 113 transições, que é quase o dobro do primeiro problema. O terceiro problema já possui 100 lugares e 209 transições. Como comparação, existem 13 problemas com menos de 100 lugares no domínio *Blocksworld*, 9 problemas no domínio *Driverlog*, 16 problemas no domínio *Logistics* e 7 problemas no domínio *Zenotravel*. No caso do domínio *Rovers*, o crescimento da complexidade dos problemas é similar à complexidade dos problemas do domínio *Satellite*.

Assim como ocorreu no domínio *Rovers* e no domínio *Driverlog*, a complexidade dos problemas fez com que o tempo de execução da busca orientadas pelas heurísticas H^1 e H^2 tivesse pior desempenho de tempo em relação ao *SatPlan* e à heurística de Töws. A heurística de Töws também foi bastante afetada pela complexidade dos problemas, pois mesmo realizando uma busca em que não se objetiva o plano ótimo, foram solucionados apenas 5 problemas.

O número de expansões da rede de ocorrências geradas no domínio *Satellite* é apresentado na figura 4.34 e o trabalho total realizado é mostrado na figura 4.35. Nestes exemplos, a relação entre o número de eventos criados e o número de expansões realizadas está na proporção de aproximadamente 10 para 1. Essa proporção não é tão grande como a encontrada em alguns problemas do domínio *Driverlog* e do domínio *Zenotravel*, domínios que tiveram índices de acerto maiores do que os problemas do domínio *Satellite*. Neste caso, a própria complexidade do problema, no que se refere ao número de lugares e transições, é que pesou mais em relação aos tempos de execução do domínio do que o trabalho total realizado pelo mesmo.

A figura 4.36 mostra o número médio de dependências do vetor de cálculo e a figura 4.37 mostra a profundidade alcançada pelas heurísticas para os problemas não solucionados.

A complexidade do vetor de cálculo acompanhou diretamente o aumento da complexidade dos problemas. Os problemas do domínio *Satellite* não estão organizados diretamente em ordem crescente de lugares e transições, como mostra a figura 4.32. A complexidade dos problemas vai subindo até o problema 17, tendo uma ligeira queda na complexidade do problema 18, subindo novamente até o último problema, que é o problema 20. Ainda que o uso das heurísticas não tenha sido eficiente para encontrar a solução para estes problemas, percebe-se que a heurística H^2 tem uma ligeira queda no seu rendimento, como mostra a figura 4.37.

4.2.7 Elevator

Neste domínio, é modelado o funcionamento de elevadores. Neste domínio, existe um elevador e diversas pessoas em diferentes andares, de forma que o objetivo é levar as pessoas aos seus destinos da maneira mais eficiente possível.

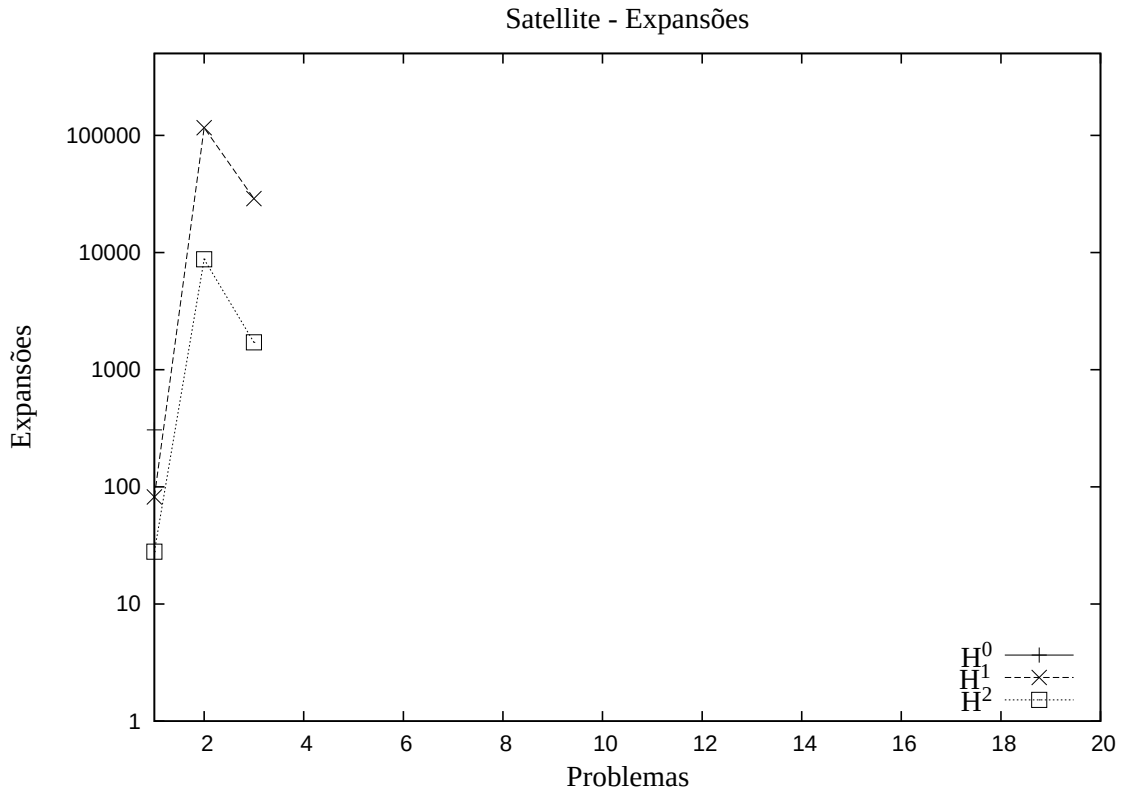


Figura 4.34: Expansões do domínio *Satellite*

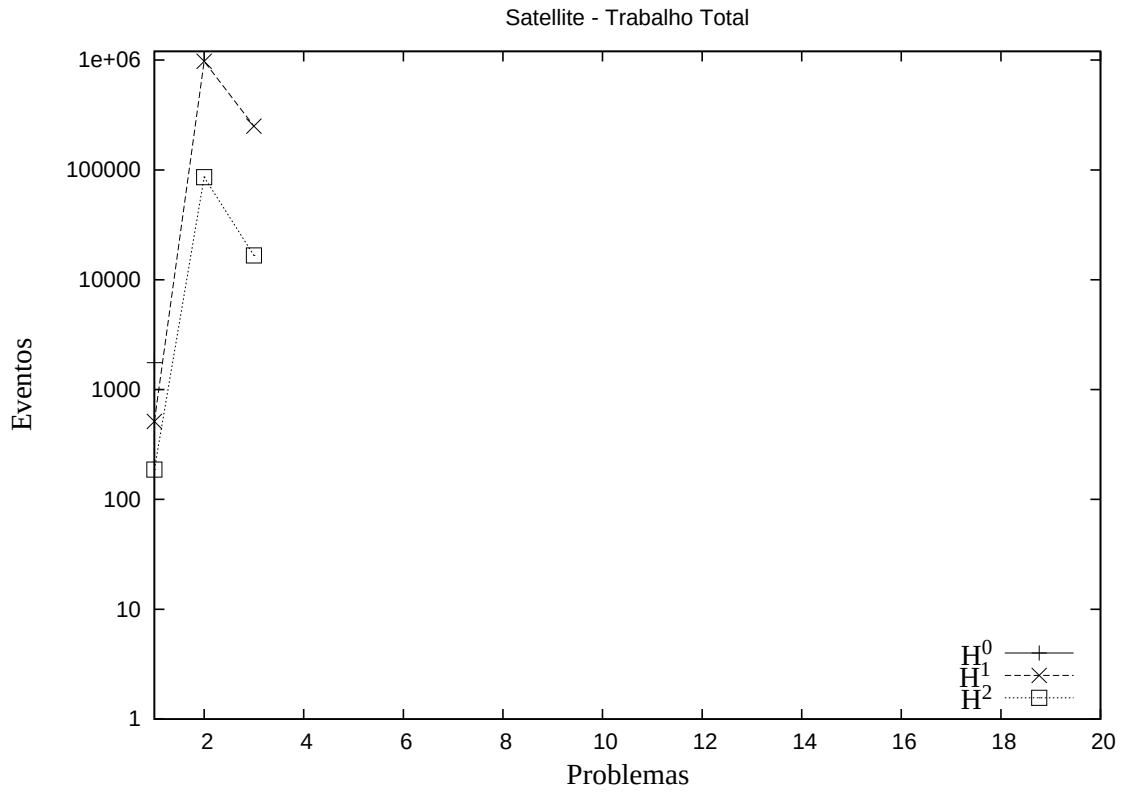


Figura 4.35: Eventos gerados no domínio *Satellite*

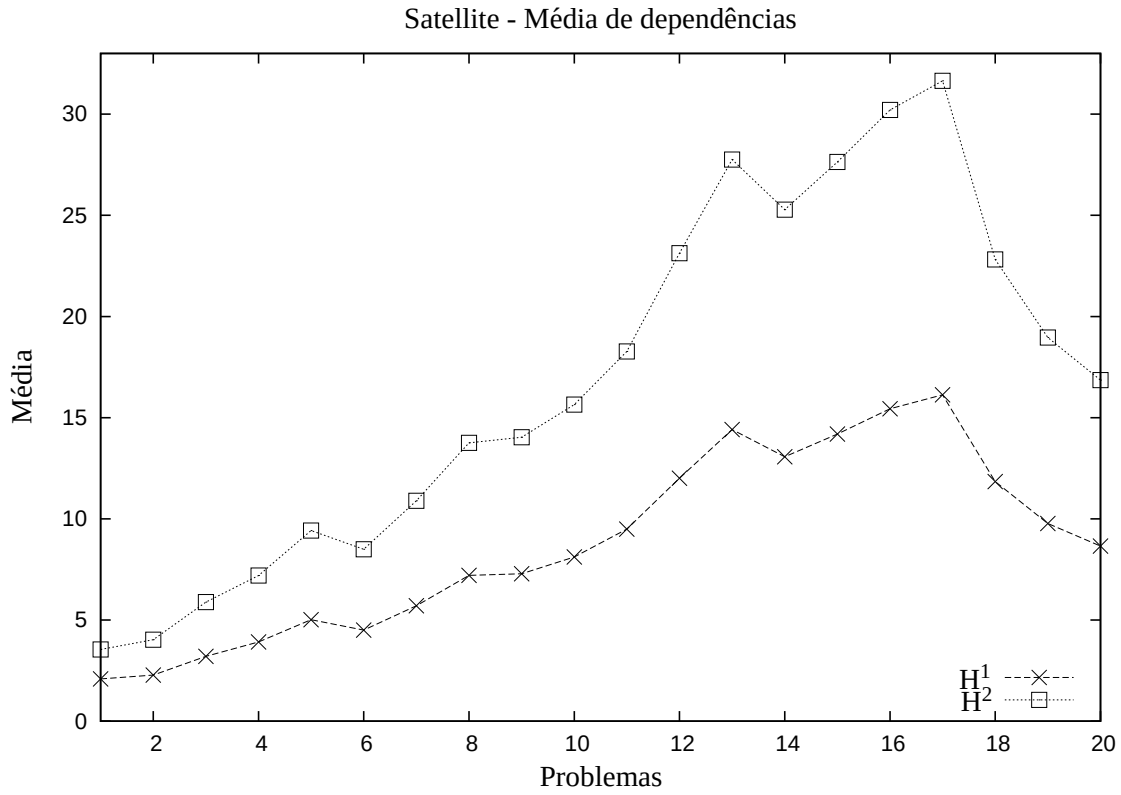


Figura 4.36: Número médio de dependências do domínio *Satellite*

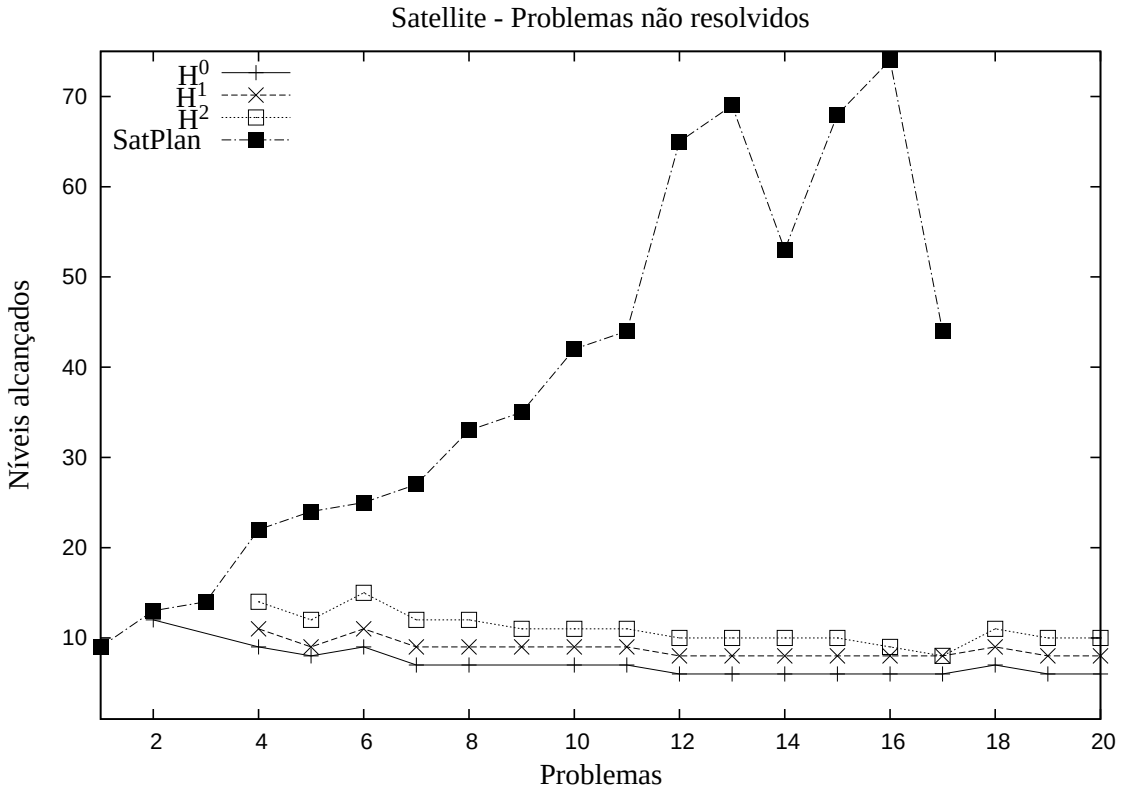


Figura 4.37: Profundidade explorada pelos problemas não-resolvidos no domínio *Satellite*

A figura 4.38 apresenta a relação entre lugares e transições dos problemas do domínio *Elevator* e a figura 4.39 apresenta os tempos de execução dos problemas do referido domínio. Como este domínio possui muitos problemas, são apresentados dois gráficos de tempos. O gráfico da figura 4.39 apresenta o tempo de execução envolvendo apenas as heurísticas H^0, H^1 e H^2 e os problemas solucionados no tempo limite de 2 500 segundos. O gráfico da figura 4.40 apresenta os mesmos dados da figura 4.39, mas engloba todos os problemas trabalhados e os tempos relativos ao *SatPlan* e à heurística de Töws.

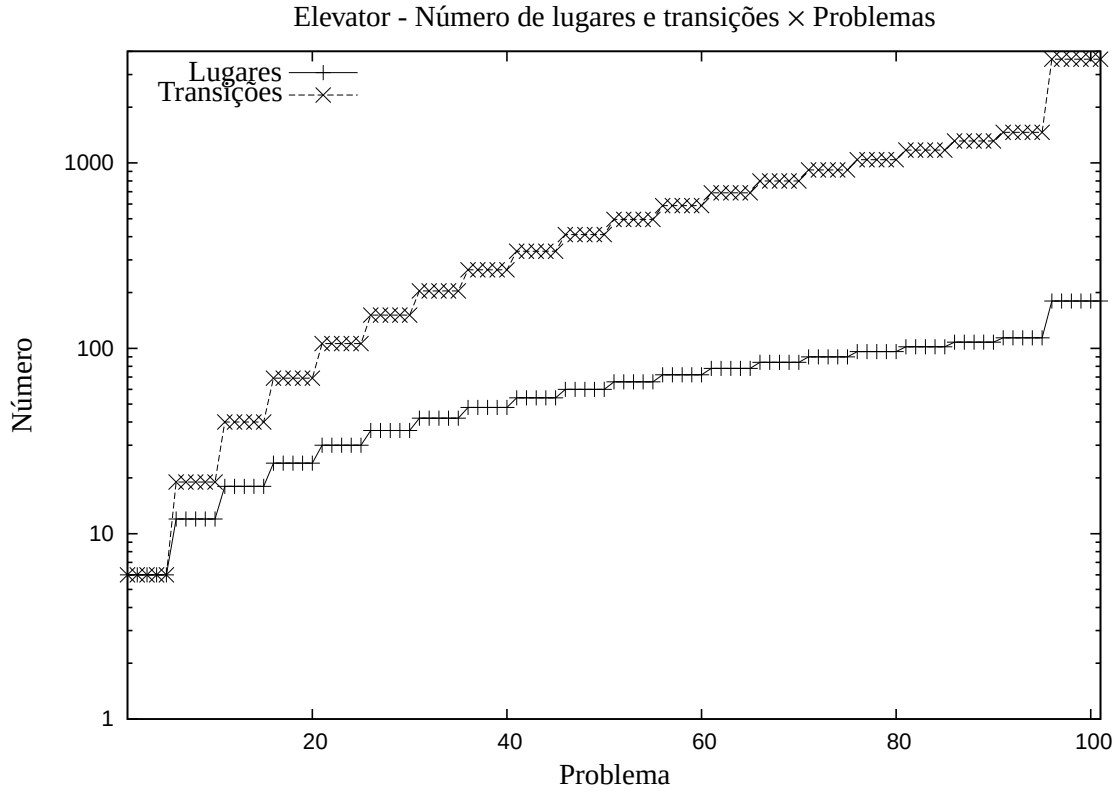


Figura 4.38: Número de Lugares e Transições das redes do domínio *Elevator*

Assim como no domínio *Logistics*, os problemas do domínio *Elevator* se caracterizam por estarem divididos em várias "categorias" de complexidade, no que diz respeito ao número de lugares e transições. Dentre todos os domínios que fizeram parte destes testes, o domínio *Elevator* é o que possui o menor número médio de lugares. Enquanto que maior problema do domínio *Blocksworld*, que possui a segunda menor média de lugares, possui 357 lugares, o maior problema do domínio *Elevator* possui 187 lugares. Já a proporção entre o número de transições e de lugares no domínio *Elevator* aumenta mais rápido do que o número de lugares. Enquanto que no primeiro problema existia uma transição para cada lugar, no último problema passaram a existir aproximadamente 20 transições para cada lugar.

Em relação aos tempos de execução, os problemas do domínio *Elevator* que foram solucionados no tempo limite de 2 500 segundos são, em sua maioria, problemas de pequeno porte. A heurística H^2 teve um desempenho pior até que a heurística H^0 em todos os problemas de 1 a

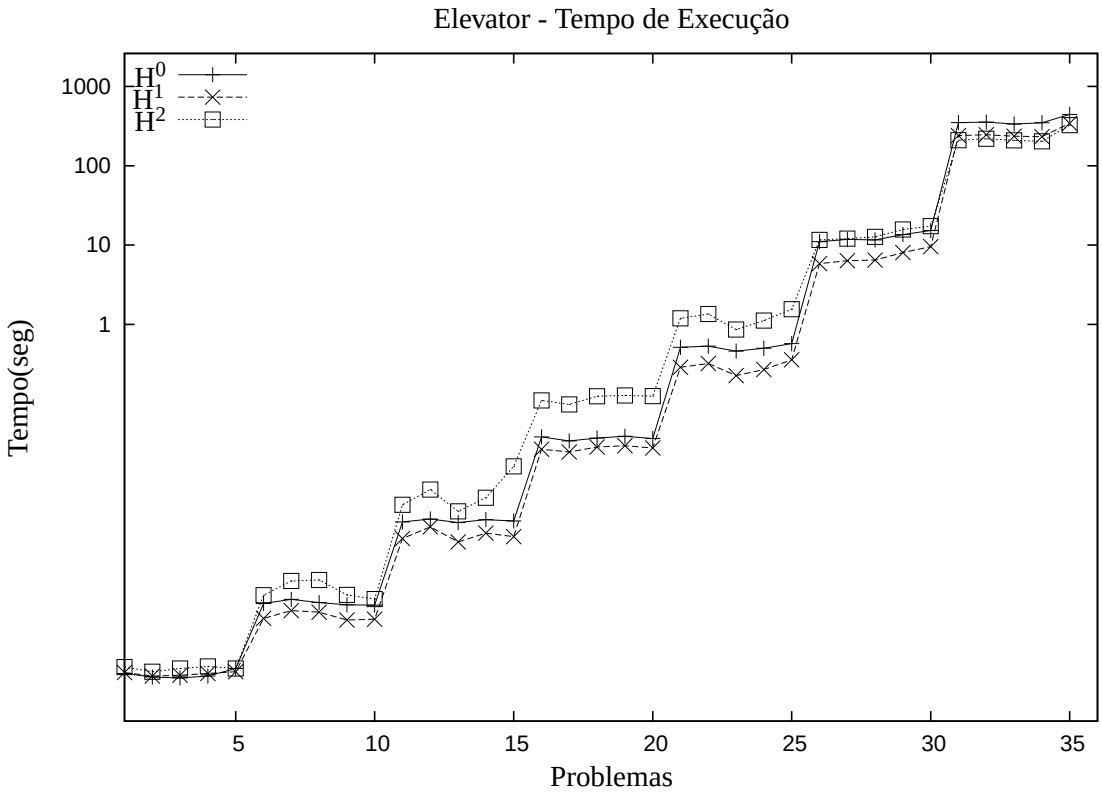


Figura 4.39: Tempo de execução do domínio *Elevator*

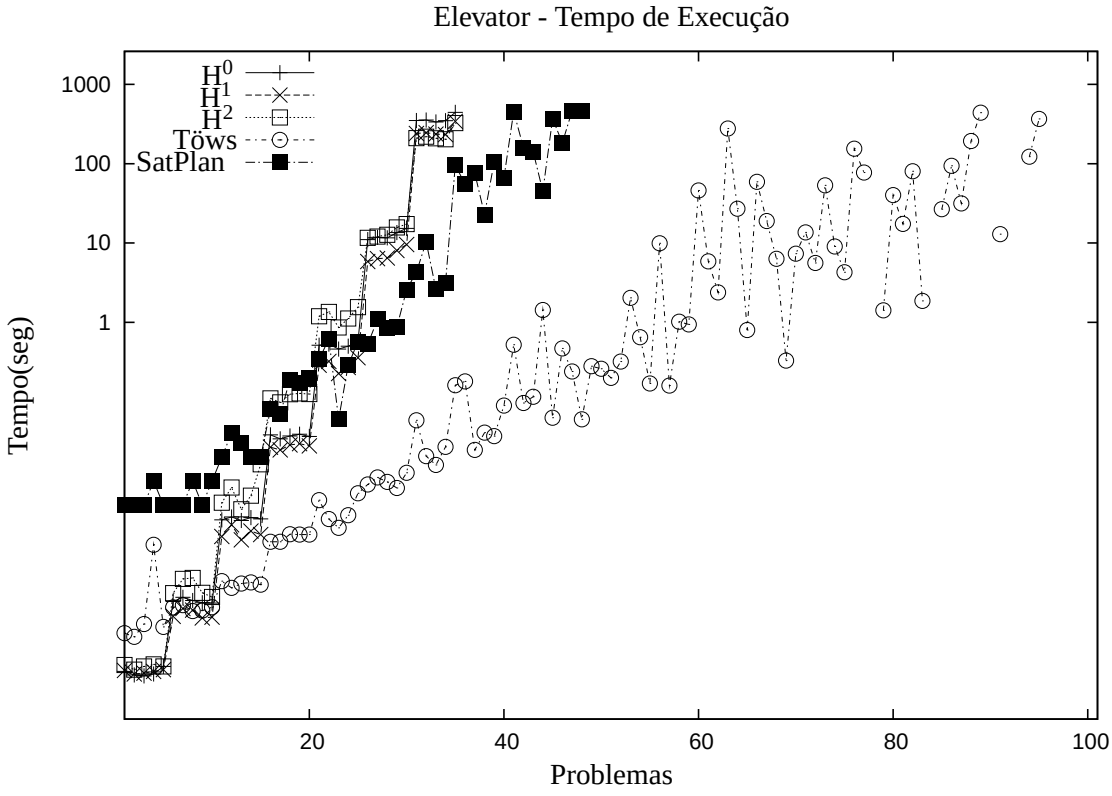


Figura 4.40: Tempo de execução do domínio *Elevator* com comparações de tempo

25, onde o tempo de execução é menor do que 1 segundo. Nos problemas de 26 a 30, o tempo de execução da heurística H^2 se tornou melhor do que a heurística H^1, mas ainda não foi mais rápido do que a busca através da heurística H^1. Isso só ocorreu nos problemas de número 31 a 35, onde o tempo da heurística H^2 foi menor do que o tempo da heurística H^1.

Da mesma forma como ocorreu em outros domínios, o tempo de execução da heurísticas H^1 e H^2 foi inferior ao tempo de execução do *SatPlan* nos exemplos de pequeno porte, perdendo em eficiência à medida em que o tamanho dos problemas cresceu. A mesma situação ocorreu quando os tempos são comparados com a heurística de Töws.

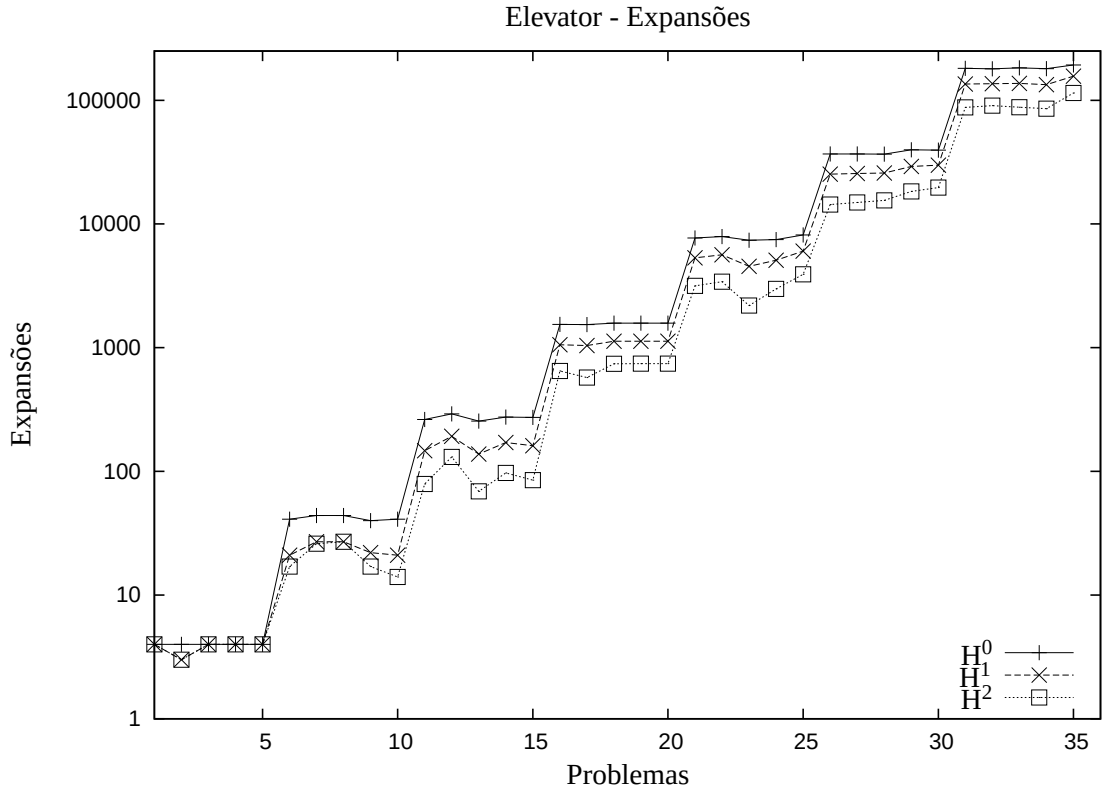


Figura 4.41: Expansões do domínio *Elevator*

A figura 4.41 apresenta o número de expansões realizadas para se obter o plano ótimo e a figura 4.42 apresenta o número total de eventos gerados durante a busca. O número de expansões da heurística H^2 é menor do que o número de expansões da heurística H^1 na maioria dos casos, com exceção dos problemas de 1 a 5, onde o número de expansões utilizando a heurística H^1 foi igual ao número de expansões realizado pela heurística H^2. O aumento progressivo da diferença entre o número de lugares e transições refletiu diretamente no trabalho total realizado pelo *Mole* nas diferentes instâncias do problema. Enquanto que nos primeiros problemas a proporção entre o número de expansões e o número de eventos criados era menor do que 2 por 1, nos problemas de 6 a 10 a proporção subiu para 3 para 1, chegando a 13 por 1 nos problemas de número 31 a 35. Esta proporção continuou subindo nos problemas que não foram solucionados no tempo limite de 2 500 segundos. A busca utilizando a heurística H^0 no problema 37 realizou 456 579

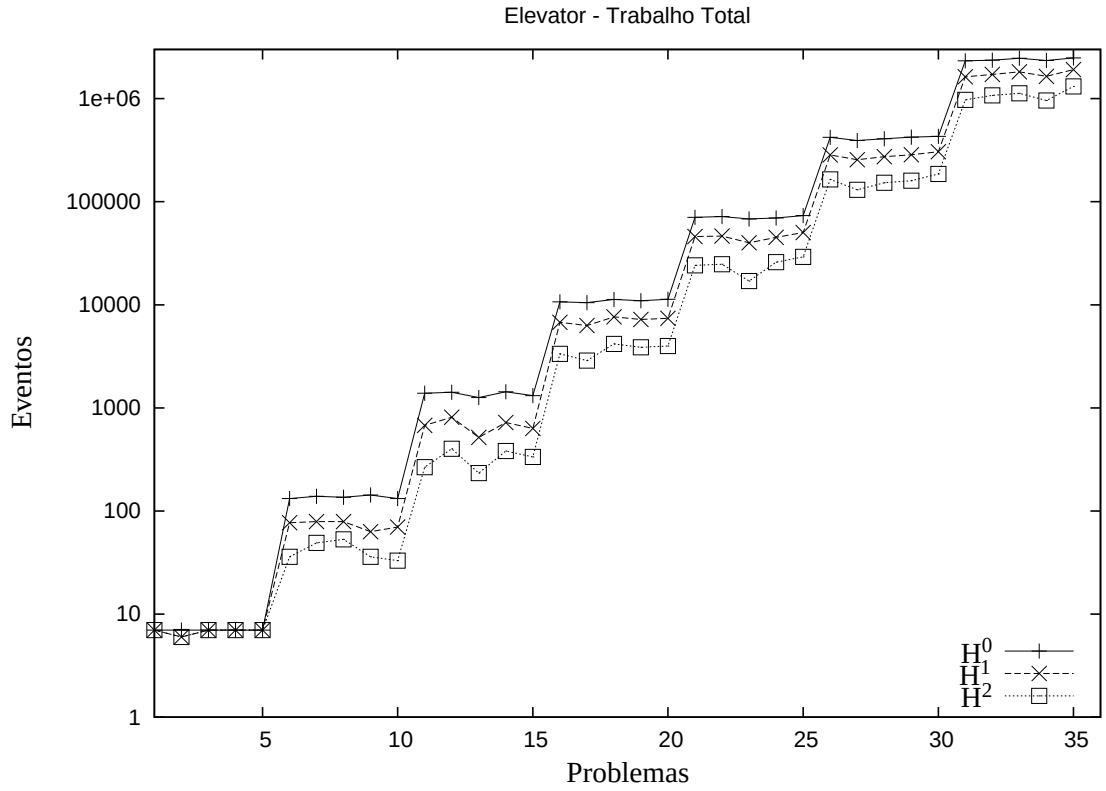


Figura 4.42: Eventos gerados no domínio *Elevator*

expansões, gerando no total 7 072 311 eventos, o que dá uma proporção de aproximadamente 15 eventos para cada expansão. Estas diferenças entre o número de eventos gerados dentre os problemas são os responsáveis pelo tempo saltar de um patamar de dezenas de segundos para centenas de uma categoria para outra, chegando em seguida em uma outra categoria em que os problemas não são resolvidos no tempo limite de 2 500 segundos.

A figura 4.43 apresenta o número médio de dependências dos vetores de cálculo gerados a partir dos problemas do domínio *Elevator* e a figura 4.44 apresenta a profundidade explorada pelos problemas não-resolvidos.

O número médio seguiu o aumento do número de transições ao longo dos domínios. O aumento da complexidade também foi acompanhado por um aumento do plano ótimo, como mostra a figura 4.44. Como ocorreu em todos os outros domínios, o aumento da complexidade dos problemas resultou em uma menor eficiência do uso de heurísticas, de maneira que a profundidade atingida começou a decair. Nos últimos 5 problemas, a execução do *Mole* foi encerrada antes dos 2 500 segundos, devido ao fato da memória alocada ter superado o limite de memória disponível. Na maior parte dos problemas, a heurística H^2 chegou mais próximo da solução do que a heurística H^1.

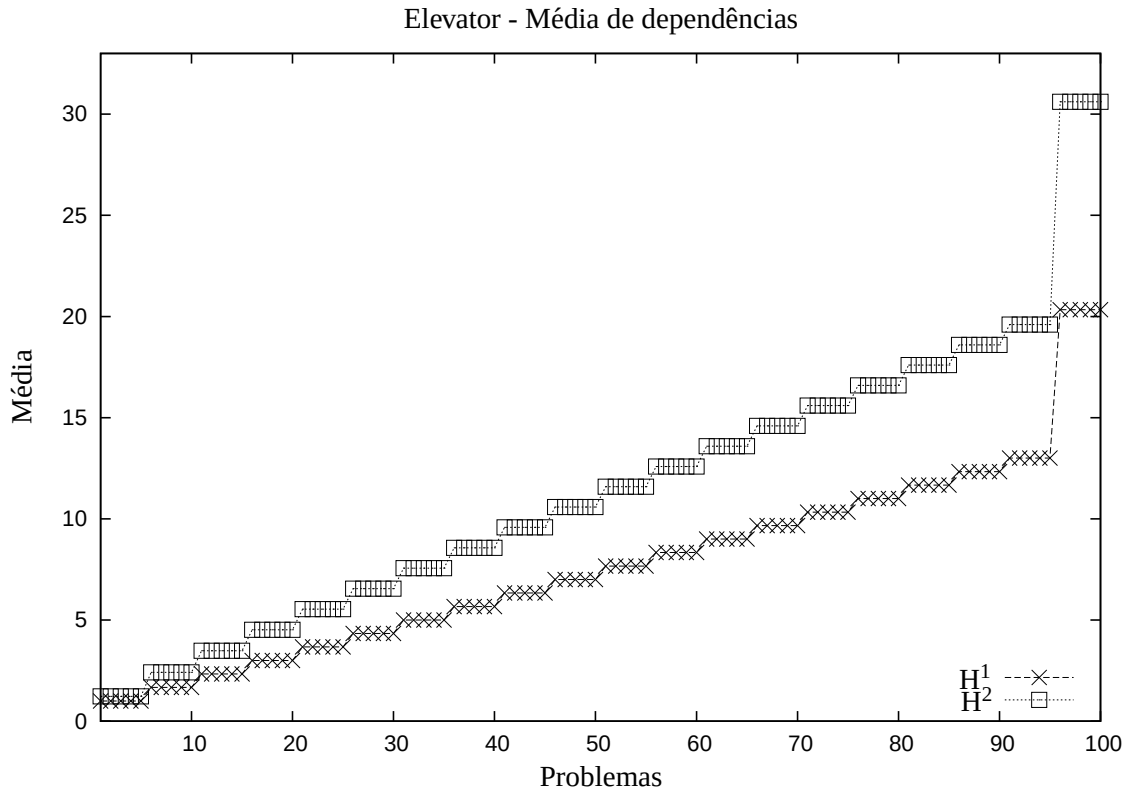


Figura 4.43: Número médio de dependências do domínio *Elevator*

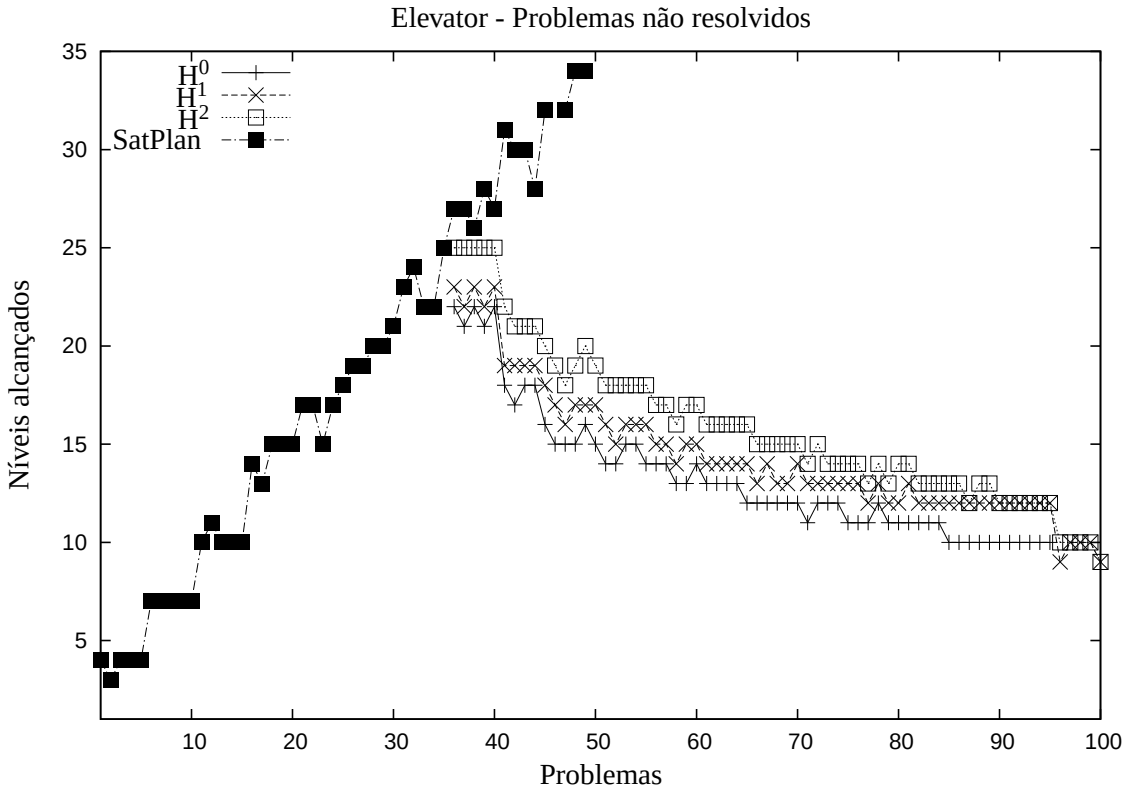


Figura 4.44: Profundidade explorada pelos problemas não-resolvidos no domínio *Elevator*

4.3 Considerações

Neste capítulo, foram apresentados resultados experimentais relativos à utilização das heurísticas H^1 e H^2 para a solução de problemas de alcançabilidade através do algoritmo de desdobramento. As redes de Petri utilizadas foram criadas a partir de problemas de planejamento pelo *Petrigraph*, apresentado na seção 2.5.2. A ferramenta *Mole* foi modificada para que a busca seja realizada através do algoritmo de busca A*, utilizando como apoio as heurísticas H^1 e H^2, apresentadas na seção 2.2.5.

As redes de Petri utilizadas nos experimentos englobam várias categorias de complexidade de problemas, sobre as quais foi imposto um limite de tempo de 2 500 segundos. As redes de Petri menores possuem, em média, 30 lugares, enquanto que as redes de Petri maiores podem conter até 900 lugares. O número de transições também varia ao longo dos domínios, abrangendo desde redes em que o número de transições é igual ao número de lugares até redes em que existem em média 38 transições para cada lugar.

O desempenho de tempo das heurísticas esteve diretamente ligado ao tamanho das redes de Petri. Nas redes pequenas, em que o tempo de execução é inferior a 1 segundo, o desempenho da heurística H^2 foi inferior ao desempenho da heurísticas H^1 e até da heurística H^0, em alguns casos. Conforme a complexidade dos problemas foi aumentando, o desempenho da heurística H^2 foi se tornando melhor em comparação ao desempenho das heurísticas H^1 e H^0. Em problemas de médio porte, foram encontrados diversos casos em que a heurística H^2 teve desempenho superior às heurísticas H^1 e H^0. Esta diferença ocorreu principalmente porque a velocidade em que ocorre a expansão da rede de ocorrências varia conforme o tamanho desta. A técnica de desdobramento associada ao uso de heurísticas não se mostrou efetiva quando aplicada à solução de problemas de alcançabilidade de grande.

Nos problemas de pequeno porte, o desempenho da busca através das heurísticas mostrou-se superior ao desempenho obtido pelo *SatPlan*, mas logo que a complexidade aumenta para problemas de médio porte, o desempenho do *SatPlan* torna-se melhor. Em comparação com a heurística desenvolvida por Töws, o desempenho das heurísticas H^1 e H^2 não é melhor em nenhum dos casos, visto que a heurística de Töws não é uma heurística admissível e a busca realizada por este baseia-se na lógica do algoritmo de busca gulosa, e não no algoritmo de busca A*.

Em todos os casos em que a solução foi encontrada por todas as heurísticas, a heurística H^2 realizou um número menor de expansões do que as heurísticas H^1 e H^0. A relação entre o número de transições e o número de lugares influenciou no número de expansões necessários para se chegar até o plano ótimo. Ao se comparar duas redes de domínios diferentes que possuem um número de lugares semelhante, mas uma grande diferença no número de transições, observou-se que as redes que possuem um número de transições maior resultaram em uma taxa

de expansões menor do que as redes de Petri semelhantes com menos transições. Isso reflete em uma demanda maior de tempo para se chegar a uma solução.

Nos casos em que o tempo limite de 2 500 segundos foi atingido, foi feita uma análise da profundidade atingida por cada uma das heurísticas. Em todos os casos, a profundidade atingida pela heurística H^2 foi maior ou igual à profundidade atingida pela heurística H^1, que por sua vez foi maior ou igual à profundidade atingida pela heurística H^0. Entretanto, o desempenho de todas as heurísticas cai conforme cresce a complexidade dos problemas.

5 *Conclusão*

Dentro do contexto das pesquisas realizadas pelo LIAMF, do DINF/UFPR, sobre a proximidade entre modelos de Planejamento Clássico e das redes de Petri, neste trabalho buscou-se a adaptação de uma heurística de planejamento para solucionar problemas de alcançabilidade de redes de Petri. As heurísticas de planejamento da família H^m foram incorporadas à ferramenta *Mole*, com o intuito de modificar o algoritmo de desdobramento de redes de Petri, de maneira que o desdobramento possa ser realizado com base na lógica do algoritmo A*, e não de acordo com o algoritmo de busca em Amplitude, tal como faz a versão original do *Mole*.

Para alcançar os objetivos deste trabalho, foram estudados os algoritmos de busca clássicos de inteligência artificial, de redes de Petri e do ambiente de planejamento. Em seguida, estudou-se com profundidade as heurísticas de planejamento da família H^m, bem como as maneiras de realizar a adaptação destas heurísticas, que possuem ligação intrínseca com a linguagem *STRIPS*, no formalismo de redes de Petri. Esta adaptação se restringiu, neste trabalho, às heurísticas H^1 e H^2. Ela foi feita a partir de uma estrutura de dados chamada de "*vetor de cálculo*", cuja função é a enumeração dos subconjuntos de lugares e suas respectivas dependências. Também foram necessárias algumas modificações na própria ferramenta *Mole* para permitir que o processo de desdobramento se comportasse de acordo com a lógica de funcionamento do algoritmo de busca A*.

Como as heurísticas da família H^m se baseiam na regressão de um estado objetivo até o estado inicial, elas são melhor indicadas para planejadores regressivos, ou seja, em que parte-se do estado objetivo até se chegar ao estado inicial. A versão modificada do *Mole* faz a busca do plano como um planejador progressivo. Como a estimativa do cálculo é feita partindo dos nós não expandidos até o estado objetivo, o cálculo precisa ser refeito cada vez que um evento é criado.

As estruturas de dados utilizadas permitiram que a regressão dos subconjuntos seja feita como pré-processamento antes da busca ser iniciada. Além disso, o vetor de cálculo permitiu que algumas otimizações fossem possíveis, como a identificação dos subconjuntos não alcançáveis a partir da marcação inicial.

Na realização dos testes, a heurística H^2 se mostrou ineficiente quando comparada à heurística H^1 e até à heurística H^0 em problemas pequenos, cujo tempo de solução é inferior a 1 segundo.

Entretanto, conforme a complexidade dos problemas aumenta, a eficiência da heurística H^2 passa a aumentar também. Em diversos problemas de médio porte, a heurística H^2 mostrou-se mais eficiente do que a heurística H^1.

A taxa de expansões da rede de Petri é uma das responsáveis pela diferença do desempenho da heurística H^2 dentre as diferentes instâncias do problema. À medida em que o tamanho da rede de ocorrências cresce, o tempo necessário para realizar uma nova expansão também cresce. Desta forma, mesmo que o cálculo da heurística H^2 seja muito mais lento do que o cálculo da heurística H^1, quando o número de expansões realizadas pela heurística H^1 é muito grande, a taxa de expansões começa a cair bastante, tornando a busca através da heurística H^2 mais atrativa. Em todos os casos de testes onde a heurística H^2 encontrou uma solução, a solução foi encontrada com menos expansões da rede de ocorrências do que na heurística H^1.

Nos problemas de grande porte, a utilização das heurísticas não foi eficiente. Em relação ao número de lugares, o maior problema resolvido foi um problema com 117 lugares. A solução não foi encontrada nestes problemas porque o aumento do número de lugares deixa o cálculo das heurísticas H^1 e principalmente da heurística H^2 mais lento, o que é associado a um aumento no trabalho total realizado pelo *Mole*, visto que os problemas maiores tentem a gerar mais eventos para cada nova expansão da rede de ocorrências. Além disso, ocorre também um aumento no tamanho do plano ótimo, obrigando o *Mole* a procurar uma solução em uma profundidade maior, utilizando um vetor de cálculo cada vez mais lento.

A implementação das heurísticas H^1 e H^2 mostrou-se menos eficiente em relação a tempo de execução do que a heurística implementada por Töws. Entretanto, isso era esperado, visto que a implementação de Töws não objetiva o plano ótimo. Em comparação com o *SatPlan*, as heurísticas tiveram um desempenho de tempo melhor nos problemas de pequeno porte, se tornando menos eficientes à medida em que o tamanho dos problemas aumenta.

Neste trabalho, foram implementadas apenas as heurísticas H^1 e H^2. Como trabalhos futuros, pretende-se implementar heurísticas de ordem maior do que 2, aproveitando-se do fato que a estrutura de dados utilizada permite que esta modificação seja realizada sem grandes modificações do código, conforme demonstra a seção 3.4. Além disso, também pode-se estudar outras formas de reduzir o tempo de busca através de otimizações do vetor de cálculo, especialmente quando a busca refere-se a heurísticas de ordem mais elevada, visto que a maior parte do tempo dispendido na busca utilizando a heurística H^2 foi o tempo do cálculo da heurística.

O conjunto de exemplos utilizados nos testes são, em sua totalidade, problemas de planejamento que foram transformados em redes de Petri, de forma que a solução pudesse ser encontrada a partir do desdobramento das redes. Entretanto, a utilização destas heurísticas para guiar o processo de desdobramento não é restrita a problemas deste tipo, visto que os valores heurísticos são calculados tomando como base apenas os elementos característicos de uma rede de Petri. Portanto, pretende-se também utilizar o desdobramento guiado pelas heurísticas H^1

e H^2 para solucionar problemas de alcançabilidade de redes de Petri que não provenham do ambiente de planejamento, como por exemplo a busca por bloqueios na rede.

Referências Bibliográficas

[1] Fahiem Bacchus. The aips '00 planning competition. *AI Magazine*, 22(3):47–56, 2001.

[2] Janette Cardoso and Robert Valette. *Redes de Petri*. Editora da UFSC, 1st edition, 1997.

[3] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence*. Addison Wesley Publishing Company, 1985.

[4] Wilkins David E. *Practical Planning - Extending classical AI Planning Paradigm*. Morgan Kaufmann, 1988.

[5] Thomas Dean, James Allen, and Yannis Aloimonos. *Artificial Intelligence, Theory and Practice*. Addison Wesley Publishing Company, 1995.

[6] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, Gordon Woodhull, Short Description, and Lucent Technologies. Graphviz — open source graph drawing tools. In *Lecture Notes in Computer Science*, pages 483–484. Springer-Verlag, 2001.

[7] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of mcmillan's unfolding algorithm. In *Formal Methods in System Design*, pages 87–106. Springer-Verlag, 1996.

[8] R. E. Fikes, N. J. Nillson, and Chris A. Cocosco. A review of strips: A new approach to the application of theorem proving to problem solving, 1998.

[9] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning - theory and practice*. Morgan Kaufmann, 1st edition, 2004.

[10] Naresh Gupta and Dana S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56:223–254, 1992.

[11] Patrik Haslum and Hector Geffner. Admissible heuristics for optimal planning. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, pages 140–149. AAAI Press, 2000.

[12] Sarah Hickmott, Jussi Rintanen, Sylvie Thiébaux, and Lang White. Planning via petri net unfolding. Technical report, In ECAI-Workshop on Model Checking and Artificial Intelligence, 2006.

[13] Richard Howey and Derek Long. VAL's Progress: The Automatic Validation Tool for PDDL2.1 used in the International Planning Competition. In *Proceedings of ICAPS'03 Workshop on the Competition: Impact, Organization, Evaluation, Benchmarks*, Trento, Italy, June 2003.

[14] Peter Jonsson and Christer Bäckström. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence*, 100:125–176, 1998.

[15] Henry Kaunts and Bart Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference of Artificial Inteligente (ECAI 92)*, pages 359–363, 1992.

[16] Derek Long and Maria Fox. The 3rd international planning competition: Results and analysis. *J. Artif. Intell. Res. (JAIR)*, 20:1–59, 2003.

[17] Paulo Romero Martins Maciel, Rafael Dueire Lins, and Paulo Roberto Freire Cunha. *Introdução às Redes de Petri e Aplicações*. IMECC-UNICAMP, 1st edition, 1996.

[18] Version This Manual, Malik Ghallab, Ecole Nationale, Constructions Aeronautiques, Craig Knoblock Isi, Keith Golden, Scott Penberthy, David E Smith, Ying Sun, Daniel Weld, and Contact Drew Mcdermott. The planning domain definition language. Technical report, 1998.

[19] K. L. Mcmillan and D. K. Probst. A technique of state space search based on unfolding. In *Formal Methods in System Design*, pages 45–65, 1992.

[20] Juliana Hoffmann Qui nhónez Benaccio and Luiz Allan Künzle. Planejamento em inteligência artificial utilizando redes de petri cíclicas. Master's thesis, Universidade Federal do Paraná, 2008.

[21] Edwin P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In Raymond, editor, *Proc. First Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 324–332, 1989.

[22] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.

[23] Wolfgang Reisig. *Petri Nets - An Introduction*. Springer-Verlag, 1985.

[24] Stuart Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach*. Prentice Hall, 2nd edition, 2003.

[25] Stefan Schwoon and Stefan Römer. Mole - an unfolder for low-level petri nets. http://www.fmi.uni-stuttgart.de/szs/tools/mole/mole-060323.tar.gz, 2004. Free software under GPL licence.

[26] Fabiano Silva, Marcos A. Castilho, and Luis Allan Künzle. Petriplan: A new algorithm for plan generation (preliminary report). In Maria Carolina Monard and Jaime Simão Sichman, editors, *IBERAMIA-SBIA*, volume 1952 of *Lecture Notes in Computer Science*, pages 86–95. Springer, 2000.

[27] Fabiano Silva, Marcos A. Castilho, and Luis Allan Künzle. A petri net based representation for planning problems. In *V International Conference on Knowledge Based Computer Systems - KBCS'04*, 2004.

[28] Christian Stehno and Ina Fc. Pep version 2.0, 2001.

[29] Guilherme Stutz Töws and Fabiano Silva. Petrigraph : um algoritmo para planejamento por desdobramento de redes de petri. Master's thesis, Universidade Federal do Paraná, 2008.